Program Analysis Based Approaches to Ensure Security and Safety of Emerging Software Platforms

by

Yunhan Jia

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2018

Doctoral Committee:

Professor Z. Morley Mao, Chair
Professor Atul Prakash
Assistant Professor Zhiyun Qian, University of California Riverside
Assistant Professor Florian Schaub

Yunhan Jia

jackjia@umich.edu

ORCID iD: 0000-0003-2809-5534

To my parents, my grandparents and Xiyu

ACKNOWLEDGEMENTS

Five years have passed since I moved into the Northwood cabin in Ann Arbor to chase my dream of obtaining a Ph.D. degree. Now, looking back from the end of this road, there are so many people I would like to thank, who are an indispensable part of this wonderful journey full of passion, love, learning, and growth.

Foremost, I would like to gratefully thank my advisor, Professor Zhuoqing Morley Mao for believing and investing in me. Her constant support was a definite factor in bringing this dissertation to its completion. Whenever I got lost or stucked in my research, she would always keep a clear big picture of things in mind and point me to the right direction. With her guidance and support over these years, I have grown from a rookie to a researcher that can independently conduct research.

Besides my advisor, I would like to thank my thesis committee, Professor Atul Prakash, Professor Zhiyun Qian, and Professor Florian Schaub for their insightful suggestions, comments, and support.

I am grateful to Jie Hui, Samsung Kwong, Alex Yoon, Kranthi Sontineini from T-Mobile, who have gaven me tremendous support during my first paper submission. I also want to thank other collaborators I have fortunately worked with, Zhi Xu, Xiao Zhang, Cong Zhen, Claud Xiao, Wenjun Hu, David Choffnes, Kevin Lau, Professor Harsha V. Madhyastha and Professor Henry Liu, for their advice and support throughout this journey.

This journey would have never been the same without my close friends Qi Alfred Chen, Yuru Roy Shao, and Ding Zhao, who always helped and pushed me through project after project. Alfred's deep love in research, Roy's Diligence, Ding's vision and interdisciplinary

knowledge helped me build my character, and added new dimensions to my work.

I'm also grateful to my fiends and colleagues Hongyi Yao, Yihua Guo, Haokun Luo, Yuanyuan Zhou, Sanae Rosen, Ashkan Nikravesh, Mehrdad Moradi, Shichang Xu, Ke David Hong, Yikai Lin, Chao Kong, Xiao Zhu, Jie You, Yulong Cao, Shengtuo Hu, Boyu Tian, Earlence Fernandes, Amir Rahmati, Dongyao Chen.

Finally, I thank my father, Xinsheng Jia, mother, Xinzhi Xue, cousin, Xuejing Lin, and my girlfriend Xiyu Hu for being the constant pillars of support in my life. Their unconditional love and support helped me bring this adventure to its end. This dissertation is dedicated to them.

I would also like to acknowledge the NSF grants CNS-1318306, CNS-1526455, CNS-1345226, CNS-1318722, CNS-1059372, CNS-1318306, and the Naval Research grant N00014-14-1-0440 for the support of my research.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Our smartphones, homes, hospitals, and automobiles are being enhanced with software that provide an unprecedentedly rich set of functionalities, which has created an enormous market for the development of software that run on almost every personal computing devices in a person's daily life, including security- and safety-critical ones. However, the software development support provided by the emerging platforms also raises security risks by allowing untrusted third-party code, which can potentially be buggy, vulnerable or even malicious to control user's device. Moreover, as the Internet-of-Things (IoT) technology is gaining vast adoptions by a wide range of industries, and is penetrating every aspects of people's life, safety risks brought by the open software development support of the emerging IoT platform (e.g., smart home) could bring more severe threat to the well-being of customers than what security vulnerabilities in mobile apps have done to a cell phone user.

To address this challenge posed on the software security in emerging domains, my dissertation focuses on the flaws, vulnerabilities and malice in the software developed for platforms in these domains. Specifically, we demonstrate that systematic program analyses of software (1) Lead to an understanding of design and implementation flaws across different platforms that can be leveraged in miscellaneous attacks or causing safety problems; (2) Lead to the development of security mechanisms that limit the potential for these threats.We contribute static and dynamic program analysis techniques for three modern platforms in emerging domains – smartphone, smart home, and autonomous vehicle. Our app analysis reveals various different vulnerabilities and design flaws on these platforms, and we propose (1) static analysis tool (OPAnalyzer) to automates the discovery of problems by

searching for vulnerable code patterns; (2) dynamic testing tool (AutoFuzzer) to efficiently produce and capture domain specific issues that are previously undefined; and (3) propose new access control mechanism (ContexIoT) to strengthen the platform's immunity to the vulnerability and malice in third-party software.

Concretely, we first study a vulnerability family caused by the open ports on mobile devices, which allows remote exploitation due to insufficient protection. We devise a tool called OPAnalyzer to perform the first systematic study of open port usage and their security implications on mobile platform, which effectively identify and characterize vulnerable open port usage at scale in popular Android apps. The analysis reveals that nearly half of the open ports are unprotected and can be directly exploited remotely, and reports vulnerabilities in over 50 popular apps to the corresponding parties for security patch. We further identify the lack of context-based access control as a main enabler for such attacks, and begin to seek for defense solution to strengthen the system security. We study the popular smart home platform, and find the existing access control mechanisms to be coarse-grand, insufficient, and undemanding. Taking lessons from previous permission systems, we propose the ContexIoT approach, a context-based permission system for IoT platform that supports third-party app development, which protects the user from vulnerability and malice in these apps through fine-grained identification of context. Finally, we design dynamic fuzzing tool, AutoFuzzer for the testing of self-driving functionalities, which demand very high code quality using improved testing practice combining the state-of-the-art fuzzing techniques with vehicular domain knowledge, and discover problems that lead to crashes in safety-critical software on emerging autonomous vehicle platform.

CHAPTER I

Introduction

Our smartphones, homes, hospitals, and automobiles are being enhanced with software that provide an unprecedentedly rich set of functionalities, which has created an enormous market for the development of software that run on almost every personal computing devices in a person's daily life, including security- and safety-critical ones. Taking the smartphone industry as an example, we can see that it took only four years for the iOS and Android, which open their software development to third party developers to dominate the market [81]. Recently, this open development fashion has quickly spread to many other emerging domains. For example, the Internet-of-Things (IoT) has quickly evolved from its initial stage, where sensors and actuators each provide hard-coded and disjoint functionality, to an application centric era, where programming frameworks are provided for third party developers to build apps to manage a single or even a number of smart devices at the same time to realize more advanced and smarter control. Many such open IoT platforms, for example Samsung SmartThings [41], Apple HomeKit [8], Google Weave/Brillo [23], and Android Things [5] have already gained great popularity among home users today.

1.1 Problems

Despite the benefit of providing an enriched set of functionalities, the open software development support of platforms also comes with security and safety risks by *allowing*

untrusted third party code to control user's device. In the mobile ecosystem, as of the end of 2016, there are more than 2 million apps available for download in Apple's App Store and Google Play, respectively. Anyone can become a developer of these apps, with a deployment fee of as low as $25, and deploy their code at a global scale. These developers can be inexperienced, careless or even malicious, posing great threats to the end users' privacy, property, and even their well-beings, if the code runs on safety-critical devices. For example, recent work [92] has shown that a malicious smart home app claiming to only monitors the battery of devices can stealthily control the user's door lock to allow break-in and theft. Moreover, we have envisioned similar threats on the emerging Autonomous Vehicle (AV), which is the next much-anticipated platform, and has unprecedented its high demand for code reliability, and strong requirement for security. As an example, inheriting from the Control Area Network (CAN) bus [17] system from traditional automotive design, self-driving functionalities are developed as individual components that plugged into the CAN bus. As the automakers usually outsource the development of some peripheral functionalities to third party suppliers, any bugs or vulnerabilities in the code could introduce great safety risks. For example, the massive recall of Jeep Cherokee in 2015 was caused by the vulnerability embedded in its 4G/LTE modem, whose compromise would put the vehicle in a situation where collisions become inevitable [24].

To limit the potential security and safety risks of running third party code, these emerging platforms usually will need to deploy a two-stage security mechanism. An *vetting* process to prevent vulnerable or malicious code from getting into the market; 2) A runtime *access control* model to restrict the software behavior, in case vulnerabilities or malice contained in the software evade the offline detection in the first stage. As a main enabler for an automated software quality assurance, the program analysis techniques including both static analysis and dynamic testing are widely adopted by the emerging platforms to build their vetting processes. Meanwhile, to ensure the runtime security and safety, a permission model is usually in place to define a software's access to sensitive resources [57]. The

program analysis technique and the permission model are the main focuses of this proposal.

However,in reality, the threats from the potentially buggy, vulnerable or even malicious software have never been thoroughly mitigated even on the most mature platform, the smartphone, not to mention the smart home and autonomous vehicle platforms, where these security mechanisms are still in the primary stage or even not in place. The reasons for these emerging platforms to remain insecure in practice are multifold, and we detail three of them from the perspectives of different parties in the software development ecosystem, which fall into the scope of this thesis.

1.1.1 Flaws being inevitable in software development and are non-trivial to detect

The security of computer systems fundamentally depends on the quality of its underlying code. Despite a long series of research in academia and industry, security vulnerabilities and flaws regularly manifest in program code, for example as failures to account for buffer boundaries [118] or as insufficient validation of input data [102]. From the app developer's perspective, it is never easy to always produce code without vulnerabilities and flaws.

The discovery of flaws and vulnerabilities is a classic yet challenging problem. Due to the inability of a program to identify non-trivial properties of another program, the generic problem of finding software vulnerabilities is undecidable [142]. As a consequence, the state-of-the-art approaches for spotting security flaws are either limited to specific types of vulnerabilities or build on tedious and manual auditing. Although some classes of vulnerabilities recurring throughout the software landscape exist for a long time, such as buffer overflows and format string vulnerabilities, automatically detecting their incarnations in specific software projects is often still not possible without significant expert knowledge [104].Thus it requires continuous efforts to extend the scope of vulnerability patterns that can be automatically detected.

1.1.2 Insufficient and coarse-grained permission system.

The permission system is concerned with limiting the access to sensitive resources on the host, and it is enforced by a reference monitor, which mediates every attempted access by a user-space program to the resources owned by the platform. From the platform's perspective, designing and implementing a permission system that balances usability and security is non-trivial. For example, web site isolation makes it difficult to share photos between two sides without manually downloading and re-uploading them. While applications can pre-negotiate data exchanges through IPC channels or other APIs, requiring every pair of applications to pre-negotiate is inefficient or impossible. From a security standpoint, existing access control mechanisms including both installation time and runtime permission systems have long been criticized to be coarse-grained, insufficient, and undemanding. For instance, they require users to make out-of-context, uninformed decisions at install time via manifests [3, 66], or they unintelligently prompt users to determine their intent [28, 53]. From numerous studies on Android and iOS permission systems [57, 146, 157, 161], a key design flaw is that the users are usually out of context.

1.1.3 Security always being an afterthought.

From the market's perspective, security have always been an afterthought in cyberspace. Many security issues, such as the ActiveX exploitation [1], the recent Spectre and Meltdown vulnerabilities [30] are all side effects of one illness: The software industry and the customers that develops the software rarely think about security first. From the software developer's perspective, functionality always has the highest priority, and in the state of affairs, little things like security is bolted on once these software are widely adopted. For example, on the emerging autonomous vehicle platforms we investigated so far, neither the security vetting nor the access control have been proposed despite its high code quality and reliability demands. It's critical to understand the unique challenges of supporting these security mechanisms on such safety-critical platforms and make them happen.

Scope	Problem	Program Analyses Applied
Smartphone	Insecure open pors leads to remote large-scale attack	Static program analysis to identify vulnerability that has predefined code patterns
Autonomous vehicle	A lack of software quality assurance measures against flaw and vulnerability	Dynamic program analysis combining guided fuzzing with domain knowledge to detect runtime problems that lead to crash
Smart home	Insufficient and coarse-grained access control	Context-based access control to mitigate the threats from vulnerable and malicious code at runtime that have evaded vetting

Table 1.1: Overview of emerging platforms covered in this proposal

1.2 Contributions

The dissertation proposes practical solutions towards addressing the aforementioned problems. More specifically, my research try to advance the security of the platforms in three emerging domains – smartphone, smart home, and self-driving car by identifying design and implementation flaws in code that lead to attacks or safety problems and build platform-level defense to mitigate the threats. Utilizing static and dynamic program analysis, I 1) systematically examined the vulnerabilities exposed by open port usage on mobile devices which lead to remote large-scale attacks; 2) designed and implemented a new access control model for appified IoT platform that enforces contextual integrity; 3) Devised enhanced fuzzing practice to test self-driving functionalities on emerging autonomous vehicle platform.

Table 1.1 shows an overview of the emerging platforms covered in this proposal and the program analyses we applied. The three proposed techniques – static program analysis, dynamic testing, and context-based access control each focus on one critical aspect of the security and safety of different platform, and are complementary to each other by limiting the threats from certain types of problems that cannot be effectively detected by the other approaches. As shown in Table 1.1, static program analysis is efficient in detecting problems at scale with code coverage guarantee, but are limited to problems whose patterns have been recognized as code properties. Dynamic testing, though does not ensure full code cov-

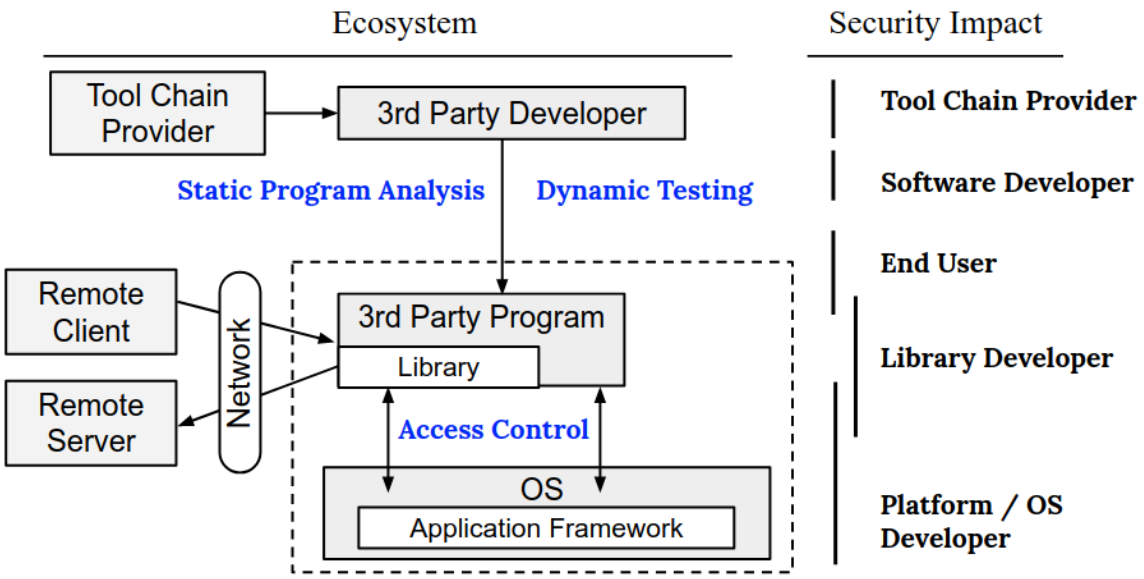


Figure 1.1: The three program analysis approaches we take in this thesis to ensure platform safety and security

erage, can be used to discover problems that are not well-defined or not distinguishable at code level, and is especially effective in detecting reliability problems in critical use cases that have real-time requirements. Shown in Figure 1.1, these two approaches can be used together to improve the vetting practice of emerging software platforms. The proposed context-based fine-grained access control, on the other hand, is responsible for restricting the suspicious program's behavior at runtime, and is complementary to the vetting process by mitigating the threats caused by the inevitable inaccuracy during the vetting.

In summary, my dissertation demonstrates that: Systematic program analyses of software (1) Lead to an understanding of design and implementation flaws across different platforms that can be leveraged in miscellaneous attacks or causing safety problems; (2) Lead to the development of security mechanisms that limit the potential for these threats.

1.2.1 Understanding and Detecting Open Port Vulnerability on Mobile Device

Open ports are typically used by server software to serve remote clients, and this usage historically leads to remote exploitation due to insufficient protection [86, 47]. Smart-

phone inherits the open port support, but since they are significantly different from traditional server machines in performance and availability guarantees, little is known about how smartphone apps use open ports and what the security implications are. We perform the first systematic study of open port usage on mobile platform and their security implications. To achieve this goal, we design and implement OPAnalyzer, a static analysis tool which can effectively identify and characterize vulnerable open port usage in Android apps.

Using OPAnalyzer, we perform extensive usage and vulnerability analysis on a dataset with over 100K Android apps. OPAnalyzer successfully classified 99 of the mobile usage of open ports into 5 distinct families, and from the output, we are able to identify several mobile-specific usage scenarios such as data sharing in physical proximity. In our subsequent vulnerability analysis, we find that nearly half of the usage is unprotected and can be directly exploited remotely. From the identified vulnerable usage, we discover 410 vulnerable apps with 956 potential exploits in total. We manually confirmed the vulnerabilities for 57 applications, including popular ones with 10 to 50 million downloads on the official market, and also an app that is pre-installed on some device models. These vulnerabilities can be exploited to cause highly-severe damage such as remotely stealing contacts, photos, and even security credentials, and also performing sensitive actions such as malware installation and malicious code execution. We have reported these vulnerabilities and also propose countermeasures and improved practices for each usage scenario.

1.2.2 Enforcing Contextual Integrity on Appified IoT Platform

The IoT has quickly evolved to the new appified stage where third party developers can write apps. Like other appified platforms, e.g., the smartphone, the permission system plays an important role in the platform security. However, design flaws in the current IoT platform permission models have been reported to expose users to significant harm such as break-ins and theft [92]. To solve these problems, a new access control model is needed for both current and future IoT platforms. We propose ContexIoT, a context-based

permission system for appified IoT platforms that provides contextual integrity by supporting fine-grained context identification for sensitive actions, and runtime prompts with rich context information to help users perform effective access control. Context definition in ContexIoT is at the inter-procedure control and data flow levels, that we show to be more comprehensive than previous context-based permission systems for the smartphone platform. ContexIoT is designed to be backward compatible and thus can be directly adopted by current IoT platforms.

We prototype ContexIoT on the Samsung SmartThings platform [41], with an automatic app patching mechanism developed to support unmodified commodity SmartThings apps. To evaluate the system's effectiveness, we perform the first extensive study of possible attacks on appified IoT platforms by reproducing reported IoT attacks and constructing new IoT attacks based on smartphone malware classes. We categorize these attacks based on lifecycle and adversary techniques, and build the first taxonomized IoT attack app dataset. Evaluating ContexIoT on this dataset, we find that it can effectively distinguish the attack context for all the tested apps. The performance evaluation on 283 commodity IoT apps shows that the app patching adds nearly negligible delay to the event triggering latency, and the permission request frequency is far below the threshold that is considered to risk user habituation or annoyance.

1.2.3 Providing Efficient Dynamic Testing Support to Self-driving Functionalities

Autonomous Vehicle (AV), which monitors the driving environment and conduct some or all of the driving tasks, has the potential to significantly change the future of ground mobility, while they must be evaluated thoroughly before the release and deployment. The recent advancement in AV has created an enormous market for the development of software-based self-driving functionalities, and ensuring the security and reliability of the code running on the AV becomes a pressing demand for the AV makers, especially when the huge code base contains third party code coming from the use of libraries, packages,

and out-sourced functionalities, such as the Mobileye vision technology [33] and Velodyne lidar technology [52]. However, after the a measurement study of several self-driving functionalities, we find that static program analysis is no longer sufficient in detecting potential risks in the code base. Some problems may not be revealed, however, until the vehicle encounters certain physical roadside conditions. Taking lessons from the fuzzing, which is one of the most effective dynamic testing approaches that are widely adopted in software testing, we propose AutoFuzzer approach, an efficient fuzz testing framework for detecting exceptions, such as crashes, failing built-in code assertions, and potential vulnerabilities specifically in self-driving scenarios.

We prototype AutoFuzzer on the Baidu Apollo [12], which is yet the most mature AV platform that are publicly available, containing complete solutions in positioning, perception, planning, control, etc. Specifically AutoFuzzer introduces an enhanced mutation engine that includes realistic atomic perturbations to improve the efficiency of fuzzing, and addresses the challenge of testing the distributed in-vehicle system by interfacing the fuzzer with the internal Inter Procedure Communication (IPC) mechanism. AutoFuzzer provides a portable solution without requiring any change to the target code bases, and is generally applicable to AV platforms with the similar Robot Operating System (ROS) [40] based architecture. The evaluation shows that it detects unique crashes and hangs much more efficient than the state-of-the-art fuzzing tool in the self-driving scenarios.

CHAPTER II

Understanding and Detecting Open Port Vulnerability on Mobile Device

2.1 Introduction

An open port (or a listening port) is a communication endpoint for accepting incoming connections in computer networking model, typically used by server applications to handle requests from remote clients. However, these ports can also be connected by malicious clients if not carefully protected, exposing potential vulnerability in the server software to remote exploitation. Such inherent weakness has always accompanied the usage of open ports throughout the history of network services, opening doors for large numbers of severe Internet attacks such as TCP SYN flooding attacks [86], the Conficker worm [47], and more recently the Heartbleed bug [25]. To mitigate the problem in these traditional usage scenarios, firewalls and user authentication mechanisms are usually adopted.

In the recent evolution to the mobile era, smartphone operating systems inherit the support for open port. But for smartphone apps, traditional open port use cases such as hosting network services no longer apply. One major reason is that compared to stationary server machines with wired network connectivity, the mobility nature of smartphones makes it difficult to maintain a stable IP address. Moreover, the IPs assigned to mobile devices are often behind a NAT (network address translation) preventing incoming network connec-

tions. Also, continuously receiving network traffic can easily drain the battery of a mobile device, leading to a form of denial-of-service (DoS) attack [164]. Due to these inherent differences, our current understanding about smartphone usage of open ports are rather limited.

With the immense popularity of smartphones, any potential smartphone open port usage may directly expose end users to severe damage. Several such examples have already been reported recently, called "Wormhole" apps [35], where open ports in popular Android apps allow an attacker to remotely collect location data, insert contacts, and even install app without authorization, and over 100M devices are affected. While these exploits are alarming, it is still unclear whether these vulnerabilities are exposed by popular use cases of open ports in the smartphone ecosystem, or just by poor implementation practices.

In this work, we perform the first systematic study of open port usage and the security implications on mobile platform. To achieve this goal, we design and implement a tool called *OPAnalyzer*, which can effectively identify and characterize vulnerable open port usage in Android apps. To use OPAnalyzer, we first formalize open port app design pattern in the language of program analysis, which in high level specifies what sensitive functions are triggered from open ports, and how they are triggered. With these definitions, OPAnalyzer first uses static taint analysis to track the information flow from the remote input entry point, and identifies the sensitive functionalities that can potentially be triggered. After this step, a set of usage paths of the open port are generated, which will lead to remote exploits if not well protected. To help prioritize human inspection, OPAnalyzer examines the security checks along the usage paths guarding the sensitive functionality. If the execution of a given path is found to have no constraints or contains only *weak* checks, a potential remote exploit is directly revealed (§2.4.4). OPAnalyzer also dynamically tests whether the vulnerable port is open by default, and labels the weak paths as *highly insecure* if the corresponding port opens automatically at app launching time. For high precision, our design leverages the Amandroid approach [156], which supports flow-, context-sensitive data-flow

analysis.

To ensure high effectiveness, we overcome several engineering challenges in the tool implementation. First, our analysis needs accurate identification of the permission-protected APIs, but the API to permission mappings provided by the most recent work, PScout [61], are incomplete for our purpose since it does not consider the prerequisites of the API usage. To address this limitation, we improve PScout to automatically fix some common missing cases (§2.4.3). Second, we find that Java reflection is commonly used to handle remote input from open ports, which is not resolved by many static taint analysis tools such as Amandroid. To ensure the call graph completeness, we add an extra analysis to locate the target class or method, which successfully resolves over 86% Java reflection use cases in our app dataset (§2.4.4). Third, we find that many apps actually implement open port usage in native code, which cannot be captured by Java-layer static analysis alone. Therefore, our tool also includes native code support based on binary analysis techniques, which is commonly excluded in nearly all existing static analysis tools on Android apps due to high engineering efforts [60, 97, 100, 156](§2.4.2).

Using OPAnalyzer, we perform an open port usage analysis on 24K popular Android apps from Google Play, and successfully classify 99% of the usage paths into 5 categories: data sharing, proxy, remote execution, VoIP call, and PhoneGap (§2.5.2). We also find that significantly different from traditional usage, ports in some categories were mostly intended only for clients in physical proximity of the smartphone, or even on the same device.

Among these open port usage families, many are found to directly enable a number of serious remote exploits if not well protected. More specifically, we use OPAnalyzer to examine the security checks along the identified usage paths, and find that they generally lack sufficient protection: for the most popular usage, data sharing, over half of the paths can be easily triggered by any remote attacker, and in some usage categories such as proxy, over 80% of the paths are not protected. From OPAnalyzer output, we uncover 410 vulnerable applications with 956 potential exploits in total, and manually confirm 57

vulnerable apps that have not been previously reported, including popular ones on the market and even a pre-installed app on some device models. These newly-discovered exploits can lead to a large number of severe security and privacy breaches. for example remotely stealing sensitive data such contacts, photos, and even security credentials and performing malicious actions such as executing arbitrary code and installing malware remotely (§2.6). To get an initial estimate on the impact of these vulnerabilities in the wild, we performed a port scanning in our campus network, and immediately found a number of mobile devices in 2 minutes which were potentially using these vulnerable apps. we have reported these vulnerabilities to the relevant parties through vulnerability tracking systems including CVE [15] and CERT [51], and some of them have been acknowledged (*e.g.,* CVE-2016-5227, VR-176). We encourage readers to view several short attack video demos at our project website [32].

Leveraging the insights from these analysis, we further categorize the vulnerable apps based on their intentions of open port, and discuss defense strategies depending on the unique characteristics in each category (§2.7). Specifically, for the physical proximity usage, which does not have any effective and usable protection yet, we propose a transparent socket-level solution that allows users to conveniently verify a connection from a device nearby and can be easily adopted by app developers.

We summarize the key contributions of this work:

- We formalize open port app design pattern, and develop OPAnalyzer to systematically characterize open port usage in Android apps and detect exposed vulnerability. To ensure high accuracy, we tackle several challenges, *e.g.,* improving the API to permission mapping completeness, resolving Java reflection, and enabling native code analysis.

- Using our tool, we perform the first systematic study of open port usage and their security implications on mobile platform. We are able to classify 99% of the identified usage into 5 distinct usage families, and discover some mobile-specific scenarios. We find that nearly half of these usage paths have no protection implemented, which can directly

be triggered by remote attackers to leak sensitive information and perform high-privileged actions.

- We perform an in-depth analysis on the vulnerable open port usage, and construct real exploits to validate the threats. From the results, we manually confirmed 57 new vulnerable apps containing popular ones on the market and also a pre-installed app on some device models, which can be used to remotely steal sensitive user data such as photos, security credentials, and perform malicious actions such as executing arbitrary code and installing malware. We also suggest countermeasures and improved practices to mitigate these problems in each intended open port usage scenario.

2.2 Background and Threat Model

In this work, we broadly define mobile apps with open TCP or UDP ports as *open port apps*. And two types of open port apps are covered by our study. (1) *Mobile service app* provides useful functionality such as sharing files on the handset by opening a file server to be connected by user's desktop. (2) *Malicious open-port apps* intentionally open ports to carry out malicious activities such as receiving commands from remote attackers for data theft or device control. Our study does not focus on malware detection, since it's very hard to distinguish malicious and legitimate open port usage without having a comprehensive understanding of the designed functionality of each app. Instead, we focus on identifying problematic usage (including both malicious and legitimate) that exposes vulnerabilities to attacker and affects the well-being of the user.

Threat model. The threat to an app with open ports comes from the attackers with the ability to reach these ports. In the design of popular smartphone operating systems such as Android, ports are reachable from both the same device, e.g., another app or a script on the web page, and another host in the same network with the victim device. Thus, compared to the majority of previously-reported smartphone app vulnerabilities that only consider the threat from on-device malware [169, 87, 90, 170, 60], open port apps additionally face

threats from network attackers, e.g., local network attacks, and web attackers, e.g., malicious scripts, which is much more diverse and also of wider range. More specifically, in this work we consider the following *three* adversary types:

(1) Malware on the same device. A malicious app, or malware, installed by the smartphone user can use `netstat` command or proc file `/proc/<pid>/net/tcp` to find the listening ports on the same device and send exploitation traffic.

(2) Local network attacker. For victims behind NAT or using private WiFi networks, attackers sharing the same local network can use ARP scanning [10] to find reachable smartphone IP addresses at first, and then launch targeted port scanning to discover vulnerable open ports.

(3) Malicious scripts on the web. When a victim user visits an attacker-controlled website using their mobile device, malicious scripts running in the handset's browser can exploit the vulnerable open ports on the device by sending network requests, which doesn't require any permission.

For each of these three threat models, we have prepared short attack video demos on our website [32] to help readers more concretely understand their practicality.

Scope and assumptions. Our study focuses on TCP ports, which are most commonly used. We did not study UDP ports, but we argue that our methodology can be easily adapted for it. Our tool is expected to handle obfuscated Android apps as long as they can be disassembled. In the current implementation, our tool only fails to analyze very few samples (0.6% of apps in our dataset); for them, even the disassembling process cannot succeed.

2.3 Design Pattern of Open Port Apps

Figure 2.1 shows a simple example Android app that opens a port for accepting remote command to push notifications on the user's device. The app first creates a `ServerSocket` to listen on a TCP port. Once a client connects to the port, the app reads the remote input

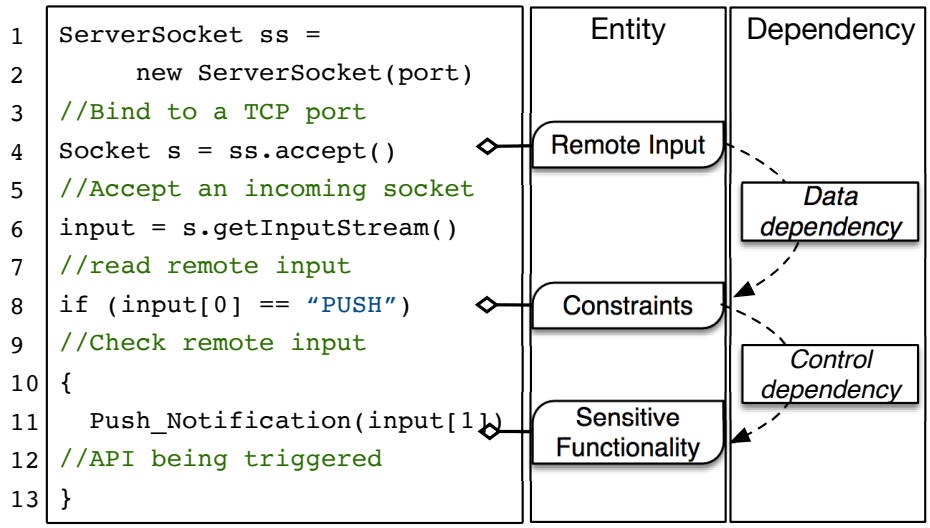


Figure 2.1: Design pattern of open port Android app

from the socket and serves the request. In this example, the app checks whether the remote input contains the "PUSH" command, and if so, it starts pushing the messages contained in the remote input to device's notification bar.

We generalize the logic of Android apps with open ports as the design pattern shown on the right side of Figure 2.1, consisting of three different entities and the dependency relationship among them.

Remote entry point is defined as the content passed to the open port app from the incoming sockets. And it serves as the entry point in our analysis framework (§2.4.1). Security mechanisms such as authentication token can be used to authorize the remote access to the app.

Sensitive functionality refers to the sensitive API set that can be triggered by remote input. The sensitive API set defined in this work contains (1) APIs protected by the Android permission system e.g., `sendTextMessage()` protected by `SEND_SMS` permission, and (2) APIs not protected by permissions but considered sensitive in the open port context. For example, an app does not require any permission to read its own data cache. However, if the data is written back to the incoming socket and transmitted to the remote client, a potential information leakage is caused, since the app cache may contain sensitive user data. We

describe our approach to construct sensitive API set in §2.4.3.

Constraints refer to the conditional statements along the path between the remote input and the sensitive APIs. It is usually introduced by the protocols that the app uses to communicate with the remote clients. If the constraints on a path are easy to bypass, the sensitive functionality on the path may be exploitable by remote attacker to launch privilege escalation attack. We discuss more on the strength of the constraints in §2.4.4.

Dependency. We identify the dependency between remote input and sensitive API as the *Program Dependency*, consisting of *control dependency* and *data dependency*. We use it to describe the "$trigger$" relationship between the source and the sink, which is shown to be efficient for us to characterize open port usage and understand their security implications.

In practice, the dependency between the remote entry point and sensitive API set of an app is not easy to characterize. The analysis must handle various program flow jumps in the Android lifecycle, including inter-component communications, Java reflection, and even jumps from native code, to accurately model the open port functionality. Moreover, to identify those vulnerable paths that can be leveraged by remote attacker, the analyzer is required to evaluate whether a given usage path is practically exploitable in terms of the timing window for attacker, and the strengths of the checks performed on the path. We design OPAnalyzer to analyze the usage and security implications of these apps to address these technical challenges.

2.4 OPAnalyzer Approach

The goal of OPAnalyzer is bi-fold (1) to characterize the open port usage on mobile devices, and (2) to identify vulnerability exposed by the usage. To achieve the goal, we design OPAnalyzer to automatically discover all the sensitive functionalities that can be triggered by remote input and examine the constraints that guard them. We define a ***usage path*** as a program path from the remote entry point to a single sensitive functionality with all the

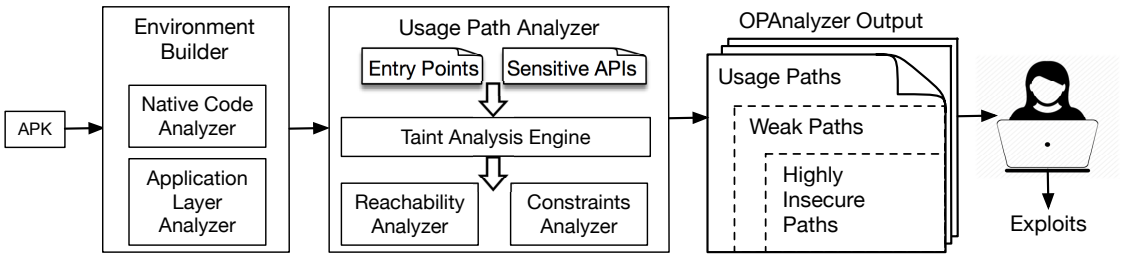


Figure 2.2: OPAnalyzer approach overview

conditional statements along the path annotated. OPAnalyzer performs the analysis on usage path level, so that various functionality of a given open port can be comprehensively examined.

Figure 2.2 shows the overview of OPAnalyzer's approach. (1) OPAnalyzer takes `apk` files as input, extracts both Dalvik bytecode and native shared objects to build the environment for the app; (2) It calculates the entry points from both the native code and Dalvik bytecode for subsequent static analysis(§2.4.1). (3) It constructs the Inter-component Data Flow Graph (IDFG) and Data Dependency Graph (DDG) from the entry points based on Amandroid, which are both flow- and context-sensitive; (4) It performs taint analysis to study the dependency between remote input and a precomputed sensitive API set(§2.4.3), and outputs the paths.(5) Constraints analyzer examines all the checks along the paths that are *control dependent* on the remote input, and annotates the strength of each constraint. (6) Reachability analyzer filters those paths unreachable from the program entry set, and annotates the run-time reachability for each path(§2.4.4). Usage categorization and vulnerability discovery are then performed on the annotated usage paths.

In the remainder of this section, we provide an overview of the main analysis steps and how we overcome several challenges.

2.4.1 Entry Point Analysis

Entry point analysis collects the remote entry points from both Dalvik and native code. It integrates `apktool` as its front-end to decompress the apk file. Dalvik bytecodes are de-

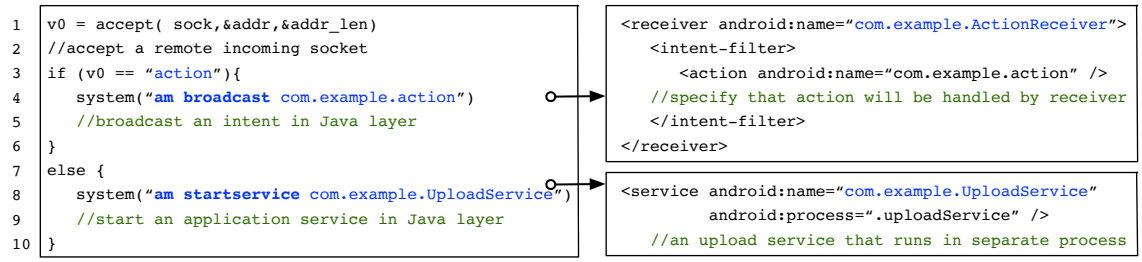


Figure 2.3: Snippet from real app showing control-flow jump from the native code layer (Left) to the application layer (Right).

coded to `smali` format and further converted to an IR called `Pilar`. The application layer analyzer extracted those Java classes that accept connections from either `ServerSocket` or `ServerSocketChannel`, which are the only Android framework APIs for apps to open TCP port in Java, as entry points. However, apps can also embed the open port functionality into the native code either for the purpose of disguising their stealth behavior or for performance reasons. Thus we implement a native code analyzer to collect those entry points embedded in native code and capture the control-flow jumps from native code to the Java layer.

2.4.2 Native Code Analyzer

Figure 2.3 is a code snippet from a real app showing the open port functionality embedded in native code. The app accepts the incoming socket in native `C` code, and passes the control-flow to application layer to serve the request. Shown on `line 4`, native code broadcasts an `intent` using the `system` function, and the `intent` is captured by the Java layer receiver and triggers the `ActionReceiver` logic. Another type of control-flow jump in this example is shown on `line 8`. The app starts a service defined in the Android manifest from native code, and the `UploadService` starts running in the background. Such cross-layer interactions cannot be captured by existing static analysis approaches, leading to inaccuracy in the security analysis. We design and implement a native code analyzer that captures such control-flow jumps based on inter-procedure taint analysis.

The native code analyzer takes the shared object files extracted from the `apk` as input, and performs taint analysis on the decompiled assembly code. The taint source is the socket accepted from the open port, while the sinks are those function calls that can initiate control-flow jump to application layer, such as `system()`. After locating the source and sink in the assembly code, it analyzes whether there is a path that propagates the taint value to the sink either explicitly or implicitly. The `system()` function calls in Figure 2.3 are not directly tainted by the `source`, but are control-dependent on the tainted conditional statement, as an example of implicit taint. The taint analysis handles inter-procedure calls, as well as asynchronous `I/O`. To handle many clients simultaneously in native code, app can perform non-blocking `I/O` using Linux's `epoll` facility [21], which provides readiness notification to simulate `I/O` multiplexing. To serve multiple requests, the thread uses `epoll_wait` to get tasks from event queue, and `epoll_ctl` to add new connections waiting to be handled into the event queue. The taint analysis propagates taint values accurately through such asynchronous function calls by keeping track of each event queue. We show how the native code analyzer improves the effectiveness of our tool in §2.4.5

The native code analyzer is implemented as a plug-in for `IDAPro` written in *Python*. It uses `IDAPro` [26] as the front-end, and performs the inter-procedural data-flow analysis on the CFG of the assembly code. The time it takes for analyzing an app depends on the number of shared object files contained in the app. The mean analyzing time is 80.0 seconds with the interquartile range of 74.9 seconds.

2.4.3 Sensitive API Selection

OPAnalyzer aims at characterizing all the sensitive functionality triggered by remote input. To define the sensitive API set and categorize their functionality, we need an accurate mapping from Android APIs to the permissions they require. However, constructing such mapping for our use case is non-trivial because our analysis is performed on the component level. We detail the challenges and our solutions below.

PScout [61] provides a static analysis approach to find the mappings between API calls and permissions. However, we find that it suffers from some completeness problems. Taking the `ServerSocket` as an example, which requires Android `Internet` permission. In the mappings generated by PScout, we only find the constructor of the `ServerSocket` mapped to the `Internet` permission, while other sensitive APIs such as `accept()` and `getOutputStream()` are missing. We suspect that it is because Android enforces some permission checks at the class level instead of API level. Although enforcing the permission check at the constructor implies that all the member functions of this class are also protected, the incompleteness of the $< API, permission >$ mappings restricts its usability for program analysis, especially when the analysis is performed on the component level. To address this problem, we design a static analysis tool to automatically add such missing APIs to the mappings. For every class constructor presented in the original PScout output, the tool takes the AOSP source code as input and extracts all the member functions of the class to complete the mappings.

In addition, we find that some APIs are not protected by Android permissions, however, when used together, are also considered sensitive in the mobile service context. For example, an app retrieves the device location and stores it in the application data cache. Once a remote connection comes in, it reads the location data from the cache and sends it to the remote attacker. In this scenario, the incoming socket does not trigger any permission check except `INTERNET`, which we think is granted by default, but the sensitive location data is stolen and leaked asynchronously. To capture such sensitive functionality in the mobile service context, we manually collect all the sources that an app can retrieve data asynchronously that do not require permission, including app cache, database, shared preference, etc. We define a pseudo permission called `DATA_LEAK`. If the data written to the incoming socket is dependent on the data retrieved from these asynchronous sources, we consider it as invoking the `DATA_LEAK` permission check. The pseudo permission together with the associated API pairs are added to the sensitive API set.

```
1  socket = ServerSocket.accept()
2  // socket = Source#1()
3  remote_input = socket.getInputStream()
4  flag = remote_input[0]
5  if (flag == "True") {
6        password = getSharedPreferences("password",0)
7        // password = Source#2()
8        outStream = socket.getOutputStream()
9        outStream.write(password)
10       // Sink()
11 }
```

Figure 2.4: An example for implicit and explicit taint logic.

2.4.4 Usage Path Analysis

To identify program dependency, OPAnalyzer performs taint analysis on the DDG rooted from remote entry points, taking the remote input as `source`, and sensitive API set as `sink`. It outputs all the usage paths that the remote input can trigger, together with all the constraints that guard the `sinks`, which are useful for open port usage categorization and vulnerability discovery.

IDFG and DDG. are built for each remote entry point using Amandroid, which include all those Inter-Component Communication (ICC) edges, and are both flow- and context-sensitive. However, the control-flow jumps introduced by Java reflection cannot be captured by this approach. Since we find that reflections are heavily used in our dataset, and also are found in those well-known *Wormhole* apps, we add reflection support to the taint analysis engine.

Java reflection is used by programs to examine or modify the runtime behavior of apps running in Java virtual machine. We have seen reflections used in apps for both legitimate purpose (e.g., bypass some API-level restrictions) and malicious purpose (e.g., disguise the entry to malicious code). Handling reflection accurately is impossible for static analysis, and remains challenging even combining runtime analysis [141]. To capture the control-flow jumps of reflection to our best effort, we implement a handler in the IDFG builder

```
1  Socket socket = ServerSocket.accept();
2  InputStream in = socket.getInputStream();
3  String input = in.readLine();
4  String command = input.split("=")[0];
5  String value = input.split("=")[1];
6  Map map = new HashMap<String,String>();
7  map.put (command,value);
8  if (map.size() > 0){                    //C1
9        if (command == "SEND_SMS") {      //C2
10             SendSMS(value);             //Sink()
11       }
12 }
```

Figure 2.5: Sample app code showing that sensitive API is protected by constraints

similar to the one used in FlowDroid [60], which can link the target class or method invoked by reflection to the calling function, and add the missing edges to the IDFG. Currently, it only handles reflection calls with targets explicitly provided in the same procedure, and will miss those whose targets are not deterministic statically. However, we find that over 86% of reflections in our dataset can be handled by applying this heuristic, and it also helps identify significantly more vulnerable paths as shown in §2.4.5.

Java layer taint analysis is used to examine the dependency among remote input and sensitive APIs. We define a notion to describe the explicit and implicit dependency relationships among statements. The notion is further used to categorize usage.

Notion 1 *The dependency relationship between two statements along the same usage path is either implicit or explicit, and transitivity applies.*

$\mathbf{stmt_1} \xrightarrow{\mathbf{E}} \mathbf{stmt_2}$: *if* $stmt_2$ *is explicitly tainted by the return value of* $stmt_1$.

$\mathbf{stmt_1} \xrightarrow{\mathbf{I}} \mathbf{stmt_2}$: *if* $stmt_2$ *is implicitly tainted by the return value of* $stmt_1$.

$\mathbf{stmt_1} \xrightarrow{\mathbf{I}} \{\mathbf{stmt_2} \xrightarrow{\mathbf{E}} \mathbf{stmt_3}\}$: *if* $stmt_3$ *is explicitly tainted by the return value of* $stmt_2$*, and both* $stmt_2$ *and* $stmt_3$ *are implicitly tainted by the return value of* $stmt_1$.

Explicit taint describes the data dependency relationship propagated by assignment ex-

pressions, while $implicit\ taint$ reflects the "triggering" relationship, in which the source is presented in the conditional statements. Shown in Figure 2.4, the first taint source comes from the remote input, and `flag` is explicitly tainted by the `remote_input`, since it is derived from it. All the statements in the `if` block are implicitly tainted by the `flag`, since they are control-dependent on the `if` statement. Specifically, another taint source is identified as the `getSharedPreferences`, which reads data from an asynchronous source. The `password` read from the shared preferences is written to the socket, identified as an $explicit\ taint$ relationship between statements in $line\ 5$ and $line\ 6$. Thus, the dependency relationship along this usage path is expressed as:

$$accept() \xrightarrow{I} \{getSharedPreferences() \xrightarrow{E} write()\}$$

To further narrow down the potential vulnerable path list, and provide insights to identify the vulnerability, OPAnalyzer integrates both constraints analysis and reachability analysis to help analyst prioritize paths output by OPAnalyzer to reduce human efforts.

Constraints analyzer examines all the conditional statements along the usage path to which the sensitive API is control dependent on. Shown in Figure 2.5, the remote input is separated into two substrings and put into a `Map`. The sensitive API `SendSMS` is control dependent on two constraints. Constraint C_1 defined in $line\ 8$ checks whether the `Map` is empty, while the constraint C_2 in $line\ 9$ checks whether the `command` passed in from remote input is "SEND_SMS". Both checks are easy to bypass, an arbitrary remote attacker can construct the input string to bypass the checks and trigger the malicious payload to be sent via SMS, which results in a remote privilege redelegation attack [90]. OPAnalyzer annotates a constraint as ***weak*** if it is either (1) comparison with constants or (2) comparison with a predefined set of trivial API (e.g., `Map.size()`, `Set.isEmpty()`). The heuristic reduces human efforts by prioritizing usage paths that are obviously easy to be controlled, but could introduce false negatives(§ 2.4.5).

Reachability analyzer characterizes the reachability of a path using both static and dynamic approaches. First, the static analysis models the app `Activity` lifecycle, and

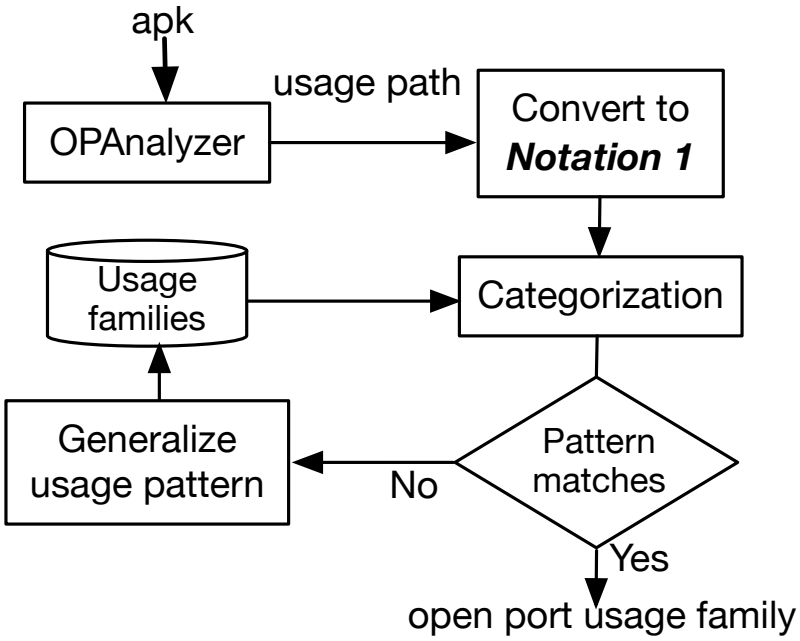


Figure 2.6: Usage categorization methodology

filters those usage paths unreachable from the program entry set. The dynamic approach identifies those usage paths that start with a port that is open by default at app launching time. These paths are annotated as ***highly insecure*** since remote attacker has a large timing window to exploit them. Note that ports that are not open by default are still vulnerable to the on-device malware in our threat model, since the malware can monitor the `proc` file and exploit the vulnerable paths as long as it detects the port is open. The dynamic analysis is implemented using function hooking based on Xposed framework [54], and combined with device automation, the analyzer automatically annotates reachability results on the usage paths.

Usage categorization methodology. Shown in Figure 2.6, OPAnalyzer aggregates usage paths with similar open port usage into one family based on the sensitive APIs they contain and the notion. It extracts usage paths and converts them to $Notion\ 1$. The categorization approach matches the notion of new paths with those already identified patterns. If an incoming path cannot be categorized into any of the existing families, we manually generalize its usage to a pattern and add a new family to the existing usage family set. Fol-

Wormhole family	Sample app	Entry points	Usage paths	Weak path		Not weak path	
				total	exploitable	total	exploitable
Baidu	BaiduMap	1	15	8	8	7	0
Tencent	QQ Downloader	4	16	1	1	15	1
AMap	Minimap	1	4	0	0	4	3

Table 2.1: OPAnalyzer output for three popular "Wormhole" apps that are reported vulnerable

lowing this approach, OPAnalyzer categorizes most of the usage paths except a few that cannot be converted to $Notion$ 1(§2.5.2).

Vulnerability discovery methodology. Considering an attacker in our threat model, to exploit a usage path and trigger the sensitive functionality, he needs to (1) find the right timing when the port is open, (2) bypass all the checks along the path to execute the sensitive API. The usage paths output from OPAnalyzer come with the reachability information, sensitive functionality, and strengths of predicates all annotated, help human analyst efficiently examine these two prerequisites for an attack, and identify vulnerability. Specifically, OPAnalyzer prioritize "weak paths" and "highly insecure" paths for manual inspection. And note that we also selectively examined the other usages paths in the OPAnalyzer output since they may also be vulnerable to remote exploits. Some of the interesting vulnerabilities (e.g., AirDroid exploit) in our case study(§2.6) are actually identified in those non-weak usage paths.

2.4.5 Evaluation

We obtain 24,000 apps from the PlayDrone dataset [152] to evaluate our tool, which contains top 1000 popular apps from each of the 24 categories in the Google Play including entertainment, tools, etc. Our evaluation focuses on the vulnerability discovery of OPAnalyzer, which is the most security critical functionality.

Discovery accuracy. To evaluate the false positives (FPs) of the weak path detection, we first define the FPs as "weak" paths that are verified to be *not* exploitable through manual inspection. We examined the weak usage paths output by OPAnalyzer, and constructed

remote input to see if the sensitive functionality could be triggered. We call these weak paths that are verified to be exploitable *vulnerable paths*, and the rest of them are considered as FPs. Among the 24,000 apps, 6.8% of them (1632) have open-port functionality, and 133 weak paths are identified to be reachable at app launching time by OPAnalyzer. We manually identified 113 vulnerable paths that can be easily exploited by constructing remote input (**FP rate 15.1%**). The FPs mainly come from paths that contain checks on the runtime property of the app. As an example, a usage path is guarded by the constraint `if (debugMode == True)` will not be triggered when the app is in release mode, but will be output falsely as weak path.

Due to the lack of any publicly accessible study of open port vulnerabilities on mobile platform, we run OPAnalyzer on the recently-reported Wormhole apps from the Chinese app market to evaluate the false negatives (FNs) of OPAnalyzer. To the best of our knowledge, they are the only reported instances that contain confirmed open port exploits, which are usable as ground truth in the FN evaluation. The FNs of the weak path detection are those paths that are not annotated as "weak", but are verified to be exploitable with our manual inspection. Wormhole apps are vulnerable due to their integrations of vulnerable SDKs including Baidu, Qihoo360, AMap, and Tencent [35], and thus we choose the most popular apps in each SDK category. We do not test on Qihoo360 library since the problem only affects an old beta version which can no longer be found in apps on major app markets.

As shown in Table 2.1, OPAnalyzer discovers usage paths that contain sensitive functionality in all of the three apps, with some of them reported as weak paths. Unfortunately, the report has no detailed path information that can be used as ground truth for evaluation, we do our best to manually analyze the decompiled app to find exploitable paths. For Baidu SDK, OPAnalyzer detects all the exploitable paths that we manually discovered, and even discovers new exploits that have not been reported in the report, such as stealing the WiFi `BSSID`. For AMap, OPAnalyzer reports four usage paths, while none of them are recognized

Feature	# of usage paths captured	Improv.	Featured app/lib
none	636	N/A	WiFi file transfer
+ reflection	804	+26%	OpenVPN
+ API	1472	+131%	AMap
+ native	845	+33%	Tencent XG
+ all	1934	+204%	Baidu wormhole

Table 2.2: Evaluation of accumulative improvement brought by (1) handling explicit Java reflection and (2) adding open-port specific sensitive API (3) capturing native code jump.

as weak paths. However, three exploits (FNs) from those non-weak paths are identified. We find that one of the conditional statement shared by all 4 usage paths depends on value that is defined beyond the IDFG of the remote entry point. OPAnalyzer thus does not consider the check as "weak" since its value cannot be determined, while it turns out to be constant in the run time. It is due to the limitation of lacking of backward analysis support, so that OPAnalyzer doesn't have enough visibility into how a variable encountered on the usage path is propagated to this procedure, if it is defined beyond the IDFG of the entry point. This affects the accuracy of the information leakage tracking and constraints analysis, and can be solved by integrating backward slicing technique [60, 163]. We plan to add that to the OPAnalyzer in the future.

False negatives may also be introduced by the native code analyzer due to a limitation inherited from the `IDA-PRO` front end. ARM processor supports multiple interaction sets mixed in one segment in the runtime, while the disassembler can interpret one type at the same time in the static analysis [9]. And when the 16-bit "Thumb" and 32-bit "ARM" instructions coexist within one segment due to optimization, the disassembling result may be incorrect. Additionally, the native code analysis of OPAnalyzer does not cover all possible types of interactions between native code and Java code. Combining native code and Java code analysis is a fundamental challenge of Android app analysis [117], and we leave it as future work to accurately model all the control- and data-flow transitions between native and Java layers.

We also evaluate the improvement on the effectiveness of OPAnalyzer's path discovery brought by three of our engineering efforts; namely (1) adding open-port specific APIs to the sensitive API set, (2) handling explicit Java reflections, and (3) capturing native code jump. Shown in Table 2.2, integrating these features greatly improves the coverage of OPAnalyzer by 204%, while completing the sensitive API set turns out to be the most effective engineering effort we spent. These improvements reduce false negatives, which is crucial to our system.

Performance. The most compute-intensive step in OPAnalyzer is the taint analysis, which includes building the IDFG and DDG, and running the Dijkstra's algorithm on the DDG to find all the usage paths. We measure the time to perform the taint analysis for the top 1000 popular apps from our dataset. The experiment runs on a machine with Intel Core i5-3470 CPU and 8GB of RAM. For apps with at least one entry point, the median processing time for OPAnalyzer to finish the taint analysis is 61.5 seconds with standard deviation of 127.2.

2.5 Usage and Vulnerability

With the usage paths output from OPAnalyzer, we systematically study open port usage and their security implications in the top 1000 popular apps from each of the 24 different categories on Google play.

2.5.1 Popularity and Permission Usage

Among the 24,000 apps, we identified open port functionality in 6.8% (1632), while 50% of these open port apps have more than 500K downloads.

Sensitive permission usage. To understand the most common functionality triggered by remote input, we use the API to permission mappings constructed in §2.4.3 to get the set of sensitive permissions. Figure 2.7 shows the top security-sensitive permissions involved in open port usage. Surprisingly, we find that in open port apps, a rich set of highly-sensitive

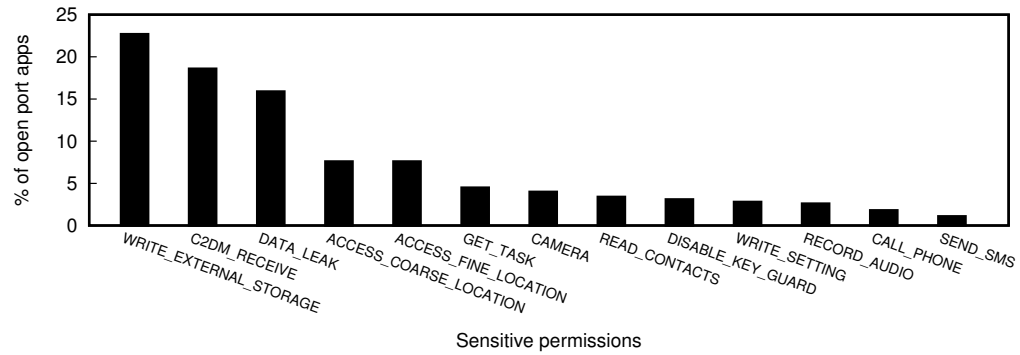


Figure 2.7: Permission protected APIs triggered by remote input.

Usage category	Pattern	Usage paths	Perc.	Apps	Weak paths	Vuln.
Data sharing	$\mathbf{accept}() \xrightarrow{\mathbf{I}} \{\mathbf{API_{data\_read}} \xrightarrow{\mathbf{E}} \mathbf{write}()\}$	1340	69.3%	425	775	V_1
Proxy	$\mathbf{accept}() \xrightarrow{\mathbf{E}} \mathbf{API_{out\_connection}}$	122	6.3%	59	101	V_1,V_3
Remote execution	$\mathbf{accept}() \xrightarrow{\mathbf{I\ or\ E}} \mathbf{API_{execution}}$	127	6.5%	41	69	V_2,V_3
VoIP call	$\mathbf{accept}() \xrightarrow{\mathbf{I}} \mathbf{API_{audio\_setting}}$	45	2.3%	27	11	V_3
PhoneGap	Categorized using code signature	282	14.6%	141	0	N/A
Uncategorized	N/A	18	0.9%	10	0	N/A

Table 2.3: Open port usage and potential vulnerability. V_1:sensitive data leakage, V_2: privileged remote execution, V_3: DoS.

OS-level functions in Android can be invoked remotely, ranging from accessing private data such as contacts and location to performing sensitive actions such as using camera and sending SMS. If not protected sufficiently, this usage can be remotely exploited to cause severe damages such as privacy leakage and privileged code executions, just like the recently reported Wormhole exploits [35]. In addition, we find that the pseudo permission `DATA_LEAK` (defined in §2.4.3), which indicates that data is read from asynchronous sources such as internal storage or content providers and sent to the remote end, has top popularity in open-port apps. This shows that open port usage commonly has potential risk of exposing internal application data to the remote attacker, which typically involves plenty of sensitive data such as credentials and conversation history [170].

2.5.2 Usage Family Categorization

Using OPAnalyzer, we categorize usage paths into different usage families defined by code patterns. Table 2.3 shows the 5 major usage families we identified in our dataset, together with the path categorization results and the types of potentially-exposed vulnerabilities. As shown, 99% of reachable usage paths of open port apps are categorized into one of the five families, which are described in detail as follows.

Data sharing path a usage path through which data read from the device is sent to the remote host. In this category, the most commonly used protocol for data sharing is HTTP, while HTTPS, FTP, UPnP [50] and some customized protocols are also observed. As shown, nearly 60% of the paths in this category are found to be weakly protected without any client authentication, leaving them easily exploitable (examples in §2.6). By examining the apps in this category, we also identified a mobile-specific open port usage scenario, *data sharing in physical proximity*, e.g., allowing a user to transfer photo to her PC nearby. This usage turns to bring most exploits in this family: as shown later in §2.6, 24 out of 26 exploits in data sharing are associated with this particular usage. Also worth noting is that in this usage family, sensitive data can be leaked without invoking any permission-protected APIs along the tainted path. Due to our improvement in sensitive API selection (§2.4.3), our tool can successfully capture these cases.

Proxy path is defined as forwarding requests in remote input to other destinations. We find that all the paths in these apps are used as local proxy, e.g., for advertising and content filtering. For example, to overcome the content modification restrictions in Android `WebView`, a web browsing app starts as background service a local web proxy so that it can insert its own ads into the fetched web pages. If exposed to remote attackers, such local proxy can be used as a reflector in targeted DDoS attacks. Also, if it is configured to cache pages or store cookies, attackers can access personalized pages to harvest user privacy, or even hijack victim's email or social network accounts for spear phishing attacks.

Remote execution path refers to usage paths that can trigger certain actions on the

Open port intention	Vulnerability description	Featured attacks	App #[1]
Usage of the app user	Lack of authentication to verify connections come from the app user	Data theft Privilege escalation	24 10
Communication with backend	Lack of authentication to verify requests from authentic app backend server	Data theft Privileged escalation	15 5
Local communication	Port used for on-device communication falsely opened to the network	DoS Data theft	12 1

[1] Number of vulnerable apps in the same category. An app may be vulnerable to both attacks in each category.

Table 2.4: Case study of verified exploitable app categorized by the intended usage of open port.

device such as sending `SMS` and writing to storage. Besides common use cases such as push notification, we also observe interesting usage in *physical proximity*, which allows the same user to use mobile device functionality through PC interface, e.g., texting `SMS` using keyboard. However, there are also sensitive functionality that can be executed remotely and are beyond the declared functionality of the app, which we suspect to be "backdoors" left by app developers.

VoIP call paths are used in apps to listen on incoming call requests based on the Session Initiation Protocol (SIP). After accepting a SIP `invite` message from the port, the app extracts information such as caller ID and starts the ring tone to notify the user. Remote attackers in theory can send spoofed packet to ring the phone and spoof the caller ID. However, due to the IPSec support in SIP, such off-path attack is unlikely to be practical.

PhoneGap paths belong to apps developed by PhoneGap/Cordova, a hybrid app development framework allowing developers to quickly build apps using JavaScript and HTML5. It uses open ports to serve requests from the JavaScript client and handle the API calls. The result written to the incoming socket is not defined in the IDFG of the remote entry point, making it difficult for OPAnalyzer to capture sensitive APIs. To address this, we use the presence of several PhoneGap-specific classes such as `CallbackServer` as the code signatures to identify these usage paths. The port intended for IPC is falsely opened to the Internet. However, we find that the usage paths of PhoneGap are protected by

strong security checks, which verifies whether the request contains a 128-bit token derived from the device's Universally Unique Identifier (UUID) [6]. Thus, we consider the open service of PhoneGap as well-protected.

2.5.3 Security Implications

Usage paths in different families, if not well protected, can lead to different security breaches. As shown in table 2.3, OPAnalyzer outputs 956 weak paths. We find that nearly half of the total usage paths are considered "weak". In the proxy category, over 80% of the paths are not protected. From these weak paths, we identify three vulnerability categories: sensitive data leakage (V_1), privileged remote execution (V_2), and DoS (V_3). Besides, we also discover a new problem that any open port on smartphones can be exploited to harvest cellular IPv6 addresses which allows attacker to collect victim IPs without scanning the huge IPv6 address space and further launch attack to exploit vulnerable ports. The vulnerability is detailed in the *Appendix A*.

V1: Sensitive data leakage. Sensitive data of a mobile device can be retrieved from many sources such as SD Card, sensor, etc. If these paths are not well protected, remote attacker can exploit them to steal sensitive data that are even protected by Android permission or *UNIX* `uid/gid` check. More importantly, if the victim IP is public, such vulnerabilities can be easily revealed using fast Internet-wide scanning tools such as ZMap [85], causing large scale data theft.

V2: Privileged remote execution. Vulnerable paths that trigger native actions can be leveraged by remote attackers to execute privileged functionality such as sending `SMS` and modifying contacts. Moreover, by exploiting the broadcasting `Intent` mechanism provided by Android, attackers can even execute functionality beyond the vulnerable app. For example, an unprotected usage path that sends `Intent` based on remote input can be leveraged to launch `YouTube` app to play the video from the URL passed by the remote attacker. By uploading a maliciously crafted `MP4` file to the URL, attacker can gain full

control of the device exploiting the *Stagefright* vulnerability [18].

V3: Denial of service. Most remote execution paths are vulnerable to DoS attack against the device user. For the local proxy usage paths, they can also be used by attackers as reflectors in targeted DDoS attack against victims in the Internet to hidden their IP addresses. Other security problems of open proxies, such as leaking internal network data and IP spoofing based attack also apply.

2.6 Exploits Case Studies

To broaden the scope of our vulnerable study, and discover more exploitable open port usage, we extended our data set to include all the 78K apps from the `Tools` category of the PlayDrone dataset to our vulnerability analysis, based on the observation that the percentage of open-port apps in the `Tools` (10.9%) is significantly higher than the average (6.8%). Furthermore, we also crawled the top 3,000 most popular apps from a Chinese app market [7], where the Wormhole problem was reported from.

By analyzing the annotated usage paths from the OPAnalyzer output, we successfully discover several new exploits of sensitive data leakage and privileged remote execution in both apps and third party libraries, including some high-profile ones with millions of downloads and even pre-installed apps. Moreover, we classify these vulnerable apps into 3 categories based on the *intended usage scenario* of the open port inferred from the manual analysis: (1) intended for use by app users; (2) intended for communication with the backend; and (3) intended for local communication. Such categorization helps better understand the challenges in securing the port opened for different purposes. Table 2.4 shows an overview of the 57 exploitable apps that are manually verified from the OPAnalyzer output. A case study of interesting vulnerability in each category is presented below. The video demos for some of the implemented attacks are shown on our website [32].

2.6.1 Intended for Use by App Users

Such apps open ports for different purposes intended for the app users, such as transferring file from the phone to another device of the same user. However, essential checks are found missing in many apps in this category, thus exposing the sensitive data and also privileged functionality of the device to attackers.

Virtual data cable is a popular app on China market that helps user transfer their photos to PC by opening a web server on the phone. The server port opens by default at app launch time and silently runs in the background. It does not authenticate clients nor notify incoming connections, thus can be easily scanned and exploited by remote attackers. Moreover, it does not check the requested file path, so that attacker can access files beyond the photo folder on SD card by adding "***../***" to the path and steal sensitive data from app cache and system directory. Similarly, a popular file sharing app *WiFi file transfer* with 10 million installs does not authenticate clients, while it opens the server port only when user presses a toggle button. However, an on-device malware that only has `Internet` permission in our threat model can listen on the status of the port by monitoring the `/proc` file system and steal data from the port as long as it is open.

PhonePal allows a user to remotely control his/her device with a web-browser, which contains highly sensitive usage paths such as open URLs in the Android browser, and open videos in the YouTube app. All these usage paths are found unprotected, which puts the device at the risk of phishing attack and even compromise [18]. Moreover, vulnerabilities such as allowing attacker to remotely install apps on the victim device, are identified in the high-profile app *AirDroid*, pre-installed on Samsung Chromebook and Smartisan phone [43]. We present a case study of this app in the mitigation strategy section (§2.7).

2.6.2 Intended for Communication with Backend

Open ports in these apps are intended to communicate with the app developer's backend server for various purposes. If the open port service does not authenticate the identity of the

remote server, attackers can spoof the app server. Interestingly, by manually examining the apps, we find that open port usage of some apps are beyond the apps' declared functionality, implying potentially covert malicious behavior.

KindeExpress is the most popular mail/package tracking app on a China market with 1 million downloads, whose functionality is to provide tracking information from many delivery service providers. One of its usage path is able to start an `Activity` of the app and display the data from remote input on the app's UI, which also brings the app to the front even when it is running in the background. We verified that none of the declared functionality of the app depends on this usage path, and we suspect it to be used for advertising. Unfortunately, this path is not protected by any authentication mechanism, and remote attacker can send command to the app pretending as it comes from the app server to display deceptive content on the app UI.

Huang CheatMaker is a game modifier app that helps users cheat when playing mobile games. We identified a usage path that accepts data pushed from the app server, stores the data as a shared object (`.so`) file and loads it at run-time. The authentication along the path is weak, and the app dynamically loads the code without verifying where it comes from nor its integrity, which enables remote attackers to inject malicious payload to the app to that will be executed, thus compromising the device.

2.6.3 Intended for Local Communication

Usage paths in Proxy and PhoneGap families are intended for on-device communication, which can be either IPC among different components of an app, or proxy for different apps on the device to use. However, open ports in some apps that are intended for local usage are falsely exposed to the network, and thus lead to security breaches.

OpenVPN is an open-source VPN implementation that is integrated by several popular VPN apps on Google Play. Multiple interfaces are provided by OpenVPN for the app to configure the local VPN settings, while the open TCP port interface is identified to be the

least secure one. A remote attacker can DoS the victim user by changing proxy settings such as port number on the device. Fortunately, some app developers paid extra attention when integrating OpenVPN, and closed the insecure configuration interface from the open TCP port, but VPN apps vulnerable to this problem still remain (e.g., *Fast secure VPN*).

CacheProxy is an app that opens local proxy with the capability of caching the web page content. Upon receiving a request, the proxy first checks whether the request can be served using cached content before fetching the page, with no authentication performed on the source of the incoming request. Remote attackers can thus easily access sensitive information of the victim, such as e-mails by requesting the e-mail page, since the page is retrieved from the cache for the attacker.

To get an initial estimate on the severity of open port vulnerabilities in the wild, we performed a small scale port scan in a subnet of a campus network. The ports scanned are those opened by the most popular vulnerable apps in our dataset, whose port number needs to be static and unique. Note that we only scanned for the existence of the open ports but did not send any data to the ports to verify the vulnerability for ethical concerns. We performed only one scan using a scanning tool [85], which finished in two minutes. Surprisingly, 40 hosts identified to be mobile devices open such ports. Although different apps that use the same port number may introduce false positives, the scanning result indicates that immediately exploitable open ports exist in the wild.

2.7 Mitigation Strategy

Traditional solutions to protect an open port from Internet attackers are through firewall, which monitors and controls incoming and outgoing traffic based on predetermined security policies. However, the firewall solution suffers from usability in the mobile context, since it is hard for individual users to configure suitable firewall rules for each app installed on the device, and coordinate both app functionality and security assurance. Moreover, in the physical proximity use scenario, since users can initiate connections from arbi-

trary hosts, it is hard to configure rules in advance.

Despite a variety of open port usage described, the fundamental problem is the lack of proper client authentications. However, we find that it is non-trivial to provide a general solution to patch the security problems for all usage cases, while preserving usability. We discuss the major challenges in different scenarios and propose countermeasures.

Intended for communication with backend. For the open port mobile app to verify that the incoming connection is from the authentic server, we suggest using secure tokens to perform authentication. Although tokens are already used in some open port apps, implementation flaws are commonly seen. For example, a third-party push notification service that distributes token to app developers for them to embed in the app is vulnerable, because the token can be extracted from the released app binary by attacker to exploit the vulnerable app installed on other victim devices. We suggest the app and server negotiate a shared token using mechanisms such as Diffie-Hellman [20] at app launching time. And open port app uses the token to authenticate further incoming connections.

Intended for use by app users. Compared to the previous usage scenario, open port apps in this case do not know in advance the legitimate remote hosts that should connect to them. Depending on the usage of the app user, the trusted remote hosts can be user's desktop or even his friend's laptop. A general solution is to use password or pin code to authenticate incoming connections; however, some security issues are raised in practice. As an example, all the apps we examined that use password, provide hard-code default password that does not require users to change before using the app, leading to the potential use of the default password and degrading app security. Randomly generated pin codes in the open port apps we examined are usually no longer than 4 characters, which is trivial to enumerate. Experiment in our local WiFi network indicates that it takes less than 5 minutes to send probing requests that enumerate all the 4-character strings.

Another authentication solution adopted by the top trending apps for this usage scenario is the incoming connection notification, which pops up a window when a new host

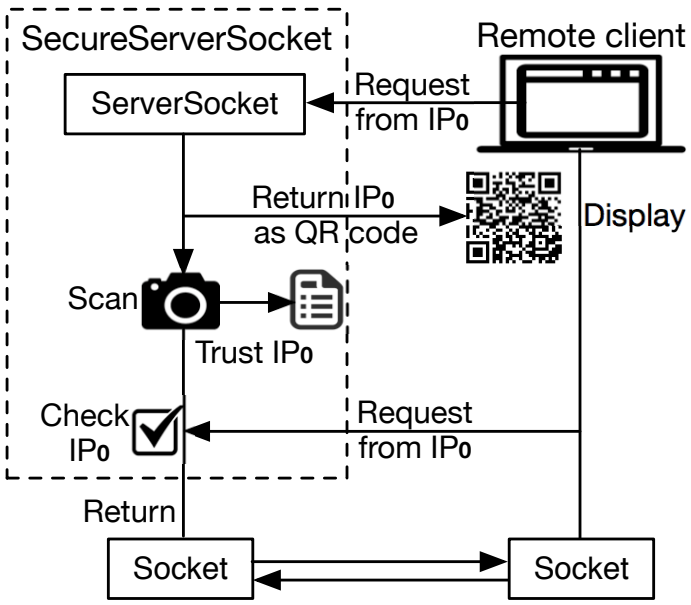


Figure 2.8: SecureServerSocket design

connects to the open port and displays the IP address of the host. And the request is not served until user explicitly accepts the client by clicking the "allow" button. For ordinary users, the timing of the pop-up window is also an important indicator for them to make the decision of whether to allow or deny the request. However, on-device malware can infer the timing when there is an incoming connection to the port by monitoring `/proc/net/[tcp][tcp6]`, and send request immediately to trick user into also allowing its connection by overlaying the pop-up window. We further find that even the most popular apps in this category has implementation flaws that can be practically exploited by attackers in our threat model.

AirDroid[1] is a top-ranked app on the market that allows users to access and manage their Android device wirelessly from desktop by opening a server on the phone. It provides a rich set of functionality to users such as access camera and install apps remotely, and uses the incoming connection notification schema to authenticate client. However, if the timing of user initiated connection is inferred by the attacker with the help of an on-device malware, and attacker sends request to the open port before the user accepts the previous

[1] AirDroid: `https://www.airdroid.com/`

connection, the app won't pop up another window. Instead, the attacker connection silently replaces the previous legitimate connection in the waiting queue, without changing the IP address displayed on the pop-up window. And when a user clicks the button to allow the legitimate connection, the attacker client is allowed instead, and numerous sensitive capabilities of AirDroid are granted to the attacker.

Usage in this category is usually for the cable-less communication with nearby hosts of the user. We confirmed that 24 out of the 26 vulnerable apps in this category are intended for the use in physical proximity. We demonstrate a transparent socket-level solution that addresses the security and usability challenges in this usage scenario. Shown in Figure 2.8, we provide `SecureServerSocket`, which encapsulates the Android `ServerSocket` API to accept incoming sockets. When a remote client, the user's desktop for example, tries to connect to the port, the `SecureServerSocket` first puts the connection on hold, and then encodes the IP address of the client into a QR code and returns to the remote client. The remote client is required to display the QR code and let the user scan it using the mobile device in order to get access.

Once the QR code is scanned, the decoded IP address is returned to the `SecureServerSocket` and the connection from this IP is allowed. It then returns the incoming socket to the upper application layer to be handled. This approach authenticates clients on the IP layer, and ensures that the open port only serves clients in physical proximity of the device user, and it achieves both security and usability. We provide `SecureServerSocket` as a library that app developers can simply use as a replacement of `ServerSocket`. No changes on the client side are required if the client is a web browser, which is the common case in our app dataset. For other client types that cannot display QR code, we suggest using one-time pin code to do the authentication instead. Demo of the SecureServerSocket implementation can also be found on our website [32]. And for future work, we plan to conduct a user study to evaluate the effectiveness of this approach.

Intended for local communication. For the local usage of on-device proxy, the best

practice is to bind the proxy to the loopback address of the device, which makes the service unreachable from the network. For another local usage scenario where different components of an app communicate using open port, we suggest using other IPC mechanisms, such as `Intent`, `Binder`, and `LocalSocket` instead. And application layer authentications are required when using them. For example, `uid/gid` check should be enabled when using `LocalSocket` to ensure that the connections are from the same app.

2.8 Related Work

Security implications of open port usage. The security implications of using open ports on the network services have been studied using Internet-wide scanning tools such as ZMap [85], revealing various vulnerabilities [86, 125, 47, 25]. However, the open port usage and security concerns on mobile platform remain under-explored. Understanding this problem in the mobile context is non-trivial, since both the current usage and the existing defense solutions are not applicable to the mobile scenario. We design and implement OPAnalyzer to bridge this gap.

Static analysis on Android. Static analysis has been used extensively in vulnerability discoveries. Specifically, on Android platform, many tools have been built to identify system vulnerability [146, 56, 119, 133, 75] and malicious apps [122, 96, 88, 73, 156, 60, 87]. Among these work, TriggerScope [96] is most closely related to our work, which focuses on detecting malicious activities embedded in narrow conditions using static analysis. OPAnalyzer serves a different goal, which is detecting open port related vulnerabilities. However, the *trigger analysis* proposed by TriggerScope can be potentially integrated by OPAnalyzer to improve its accuracy of weak constraints analysis. FlowDroid [60] and Amandroid [156] are two static analysis tools similar to OPAnalyzer, both of which model the Android `Activity` lifecycle [60], and capture inter-component communications. Compared to these generic app analysis tools that evaluate the app as a whole, our tool focuses on examining the part that contains open port functionality. Another difference is that we

integrate native code analysis for high accuracy and coverage of our analysis, which is commonly excluded in these tools due to high engineering effort.

Android app security. Mobile app security issues have gained much attention recently, and research efforts were made on detecting repackaged apps [109, 73, 57, 169], apps with malicious behavior [101, 171, 96, 161, 77, 91], or apps with vulnerability [111, 90, 79]. Different from these prior studies, we investigate the vulnerability in open port apps, which is not covered by related work, and has a new threat model not previously explored. Our analysis results shed important light on the common design and implementation flaws in these apps, and we also propose solutions to some mobile-specific usage scenarios.

2.9 Conclusion

In this work, we develop a tool called OPAnalyzer, which can systematically characterize open port usage in Android apps and effectively detect exploitable vulnerabilities. Using this tool on 24K popular Android apps, we are able to classify 99% of the mobile usage into 5 families, and identify some unique usage scenarios on mobile platform. From the vulnerability analysis performed, we find that such usage is generally unprotected. We are able to discover a bunch of new exploits causing vulnerabilities such as information leakage, denial of service, and privileged execution.

We also propose countermeasures and improved practices to mitigate these problems in different usage scenarios. As a potential future work, we want to apply OPAnalyzer to analyze Android system applications to discover more critical vulnerabilities.

CHAPTER III

Enforcing Contextual Integrity on Appified IoT Platform

3.1 Introduction

The Internet-of-Things (IoT) has quickly evolved from its initial stage where sensors and actuators each provide hard-coded and disjoint functionality, to a new *appified* era, where programming frameworks are provided for third-party developers to build applications (apps) to manage a single or even a number of smart devices at the same time to realize more advanced and smarter control. Many such appified IoT platforms, for example Samsung SmartThings [41], Apple HomeKit [8], and Google Weave/Brillo [23], have already gained great popularity among home users today.

Like other appified platforms such as the smartphone platform, the permission model plays an important role in the security of these appified IoT platforms, defining an app's access to sensitive resources [57]. However, security-critical design flaws in the permission[1] model of these platforms, for example *overprivilege* problems due to the current coarse-grained permission definitions, have already been reported recently, exposing smart home users to significant harm such as break-ins and theft [92]. To solve these problems, a new access control model is needed in these appified IoT platforms in order to provide home users with more fine-grained control of app behavior.

Existing access control mechanisms employed by the most recent appified platform

[1] SmartThings uses the term `capability` instead of `permission` [92]

with huge popularity—the smartphone platform—have long been criticized to be coarse-grained, insufficient, and undemanding [57, 143, 157, 160], which are quite similar to the aforementioned problems in current IoT platforms. From numerous studies on Android and iOS permission systems, a key design flaw is that they either require users to make uninformed decision at install time [3] or prompt users at runtime when an app requests any of a handful of resources, without providing essential contextual information [4, 28]. These studies conclude that it is highly desirable to put the user in context when making permission granting decisions at runtime. This helps ensure a property known as "contextual integrity" defined by Nissenbaum [132] with which "information flows according to contextual norms," and it is advocated as the desired norm for future permission system design of the smartphone platform [157, 161, 57].

Taking lessons from previous permission systems, this work aims to provide contextual integrity in appified IoT platforms in order to solve the security problems arising in current IoT platform permission systems. However, as discussed in previous attempts to support it in the smartphone platform [157, 57], providing contextual integrity in appified systems is challenging due to two reasons:

- **Availability of context** is not guaranteed in the lifecycle of the app: majority of sensitive permission requests occur when the user is not interacting with the requesting app [157]. This situation only gets worse in the IoT scenario, since unlike the UI-oriented smartphone apps, the whole point of developing IoT apps is to provide automated device control with minimum user involvement. Except sending notifications, usually no user interaction is required after the app setup procedure, making it more difficult to involve user in the context at runtime.

- **Frequency of prompts** is another important factor for a permission system to be effective [157]. On the smartphone platform, since the request frequencies for some permissions are too high to prompt the user each time a request occurs without risking user habituation or annoyance, current designs shift toward a model of only prompt-

ing the user the first time a request occurs to increase usability [157]. However, this harms contextual integrity since the subsequent sensitive actions may be performed in a completely different context than that of the initial request.

In view of these challenges, we design and implement *ContexIoT* (means putting IoT into context), a context-based permission system for appified IoT platforms which supports fine-grained identification of context for a sensitive action and runtime prompts with rich context information to help provide contextual integrity. In our design, the context is defined at inter-procedure control and data flow levels, and can be flexibly tuned to support different context granularity in order to best balance security and usability. ContexIoT is designed to be backward compatible, and thus can be directly adopted by current IoT platforms to provide more effective access control.

At a high level, ContexIoT design is based on the observation that a permission granted by the user is expected to allow the triggered app functionality *only under that particular usage context*. We abstract the usage context of an app functionality as a program path, and thus define the context as the execution flow of the code at runtime, including how the functionality is triggered and what data is flowing along the execution path. This definition falls into the trigger-action based programming model of IoT apps [151], so that when the user is prompted, the context can be naturally represented as the triggering sequence of real-world physical events. To help the user make a more informed decision, we use taint analysis to track the runtime data on the execution path and label the data source when presenting the context information to the user, e.g., showing whether the data to be sent out is the user password or just the battery level. We compare ContexIoT context definition with existing context-based security approaches for smartphone platforms, and find that our fine-grained definition at inter-procedural control and data flow levels can successfully identify stealthy attack paths that can evade other systems, showing better visibility than previous design.

We built a prototype of ContexIoT on the Samsung SmartThings platform, which at

the time of writing has the largest number of supported device types and IoT apps (called SmartApps) among all the IoT platforms [92]. To support existing SmartApps without changing the closed-source SmartThings cloud backend, we developed an app patching mechanism that can convert unmodified commodity SmartApps to ContexIoT-compatible SmartApps. The patching process separates the execution flow of a sensitive action in the original SmartApp into two steps: (1) Collect the context information before the action is executed, and (2) Allow or deny the action based on the in-context user decision. ContexIoT uses a cloud backend to remember the previous decisions by maintaining a mapping between an in-context sensitive action for an app and the granting decision. If no mapping is found, the system prompts the user with the context and the requested action, and stores the user decision to the ContexIoT cloud backend.

To evaluate the effectiveness of our approach, we extensively collect the reported IoT attacks from multiple sources. For exploits on non-appified platforms, we explore the possibility of migrating them to appified IoT platforms. In total, we have constructed 10 SmartApps that are either malware or vulnerable apps based on the reported IoT attacks. Considering that appified IoT platforms are still in a primitive stage and not many attacks are reported, we further survey malware classes from appified smartphone platforms. We taxonomize them into 4 categories based on the malware lifecycle, with 3–6 species in each category. Out of the 17 species in total, we find that 15 of them can be naturally migrated to IoT platforms due to the similarity of appified platforms. Overall, we build an IoT attack app dataset with 25 SmartApps, each representing a unique attack class. Evaluating ContexIoT on this dataset, we find that all 25 different attack execution paths have been successfully distinguished with the context information correctness manually confirmed.

For performance evaluation, we build a dynamic testing framework based on the device simulator provided by the SmartThings IDE. Using this framework, we dynamically inject virtual device events and are able to trigger all the 916 event handling logic in 283 SmartApps. From the performance measurement results, we find that the SmartApp patching

logic only introduces 67.1 ms additional delay on average, which is negligible in practice since the end-to-end latency is dominated by the network latency between the SmartThings cloud backend and the physical device. We also evaluate the frequency of prompts, and find that the average possible life-time of permission request prompts is only 3.5 times for each SmartApp on average, which is far below the threshold that is considered to risk user habituation or annoyance [160, 157].

To summarize, our contributions in this work are three-fold:

- To understand the design requirements for a context-based permission system on IoT platforms, we perform the first extensive study of possible attacks on appified IoT platforms by reproducing reported IoT attacks and constructing new IoT attacks based on smartphone malware classes. We categorize these attacks based on the lifecycle and adversary techniques, and build the first taxonomized IoT attack app dataset with 25 SmartApps, each representing a unique attack class.

- We design and implement ContexIoT, a context-based permission system for appified IoT platforms that supports fine-grained context identification and rich context information prompting at runtime to help provide contextual integrity. To distinguish fine-grained context, ContexIoT defines context as execution paths at inter-procedure control and data flow levels, which is shown to be more comprehensive than previous designs for smartphone platforms. To help users make more informed decisions, ContexIoT also labels the data source of the runtime data using taint analysis. To provide backward compatibility, ContexIoT contributes an app-patching mechanism that converts existing IoT apps to ContexIoT-compatible apps.

- We prototype ContexIoT on the Samsung SmartThings platform, and evaluate it on our IoT attack app dataset for system effectiveness with over 283 existing SmartApps for system performance. For the attack app dataset, we find that all attack execution paths have been successfully distinguished with correct context information anno-

tated. The performance evaluation results indicate that ContexIoT app patching adds nearly negligible delay, and the permission request frequency is far below the threshold that is considered to risk user habituation or annoyance.

3.2 Related Work and Background

In this section, we cover previous work on permission-based access access control and IoT security and necessary background for Samsung SmartThings platform, and we clarify the goal and the problem scope of this work.

3.2.0.1 Permission-based Access Control

The permission-based access control plays an important role in the security of appified platforms, and has received a lot of attention by the security research community [87, 61, 128, 160]. Acar *et al.* pointed out that the current concept of permission granting mechanism has failed in practice, and proposed a clean break to seek for permission revolutions [57]. Backes *et al.* advocates *contextual integrity* as the desired norm for future permission systems design based on a rigorous user study on Android platform [157]. Roesner *et al.* introduced User-Driven Access Control where the user is kept involved with access control decisions in case-by-case basis by using access control gadgers [143]. Rahmati *et al.* introduced the concept of context-specific access control [139] in Android where app Activities are used to distinguish different user contexts. Compared to these previous systems for the smartphone platforms, this paper aims to provide contextual integrity to the IoT platforms, which faces several IoT specific challenges, e.g., it is more difficult to involve users in the context. Also, the context in ContexIoT is defined at both control and data flow levels, which is more fine-grained. We compare our context definition with those in these previous work later in §3.5, and it shows that our definition is more comprehensive and can defeat attacks that can evade these previous design.

Another line of research focuses on improving the usability of the permission system.

Felt *et al.* introduced a set of guidelines on when and how to request permissions [89], which can instruct the design of default security policies. Wijesekera *et al.* suggested the systems to learn about their users' privacy preferences and only confront users with consent dialogs when a permission request is unexpected for the user [157]. Keley *et al.* proposed to enrich permission dialogs with more detailed privacy-related information to help users make more effective decision [114]. In comparison, ContexIoT targets an orthogonal goal, i.e., enabling effective identification of fine-grained context for security sensitive actions. Leveraging the rich context information collected in ContexIoT, these approaches can be combined with ContexIoT to improve usability.

3.2.1 IoT Security

The IoT security research is centered around three themes: Devices, Protocols and Platforms. In the IoT device scope, many Telnet-capable IoT devices are reported to be vulnerable due to weak/default password or unprotected debugging interfaces [162]. Ur *et al.* identified problems in the access control of the Philips Hue lighting system and the Kwikset door lock that fails to enable essential use cases [150]. Ronen *et al.* demonstrated extended functionality attacks on smart lights that can leak information and causing seizures using strobed light [144].

On the protocol level, researchers demonstrated flaws in the ZigBee and ZWave protocol implementations of IoT devices [94]. More recently, the misusing of some protocols in some IoT specific scenarios has been reported to cause security and safety problems [105]. For example, using the BLE (Bluetooth Low Energy) range as the proof to verify physical proximity is considered insecure in the auto-unlock usage scenario. In our work, we extensively survey these IoT attacks, and explore the feasibility of migrating them to the appified IoT platform.

On the IoT platform level, recent work discovered a series of security-critical design flaws such as the coarse-grained permission definition on the SmartThings platform [92].

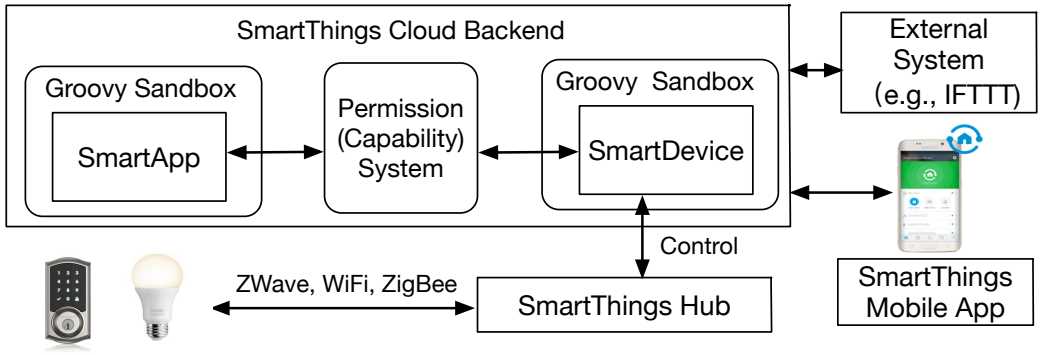


Figure 3.1: SmartThings architecture overview

To limit the usage of sensitive data, Fernandes *et al.* proposed the FlowFence framework [93] that supports flow policy rules for IoT apps. Our work is similarly motivated by the security problems in the appified IoT platforms. However, unlike FlowFence, our approach does not require additional developer effort and is backward compatible. Moreover, ContexIoT allows user control in cases where a particular data flow might be allowed in one scenario, but should be blocked in another.

3.2.2 Background

In this work, we focus on the Samsung SmartThings platform, which uses a popular cloud-backed architecture design as shown in Figure 3.1. Other popular IoT platforms such as Apple's HomeKit and Google's Weave/Brillo also use such design, and the differences only lie in the communication protocols used in the wireless hop. As shown later in §3.5, ContexIoT also leverages such cloud-backed architecture, and thus is generally applicable to these popular IoT platforms today.

As shown in Figure 3.1, the SmartThings ecosystem consists of three major components: a hub, a SmartThings cloud backend, and a smartphone Companion App. The IoT apps in the SmartThings platform are called *SmartApps*, which are written in Groovy using the Web based IDE provided by SmartThings. These SmartApps are not running on the IoT devices. Instead, they are executed by the SmartThings cloud platform within a sandboxed

environment. The sandbox is an implementation of a Groovy source code transformation that only allows whitelisted method calls to succeed in the SmartApp, and thus disables some object-oriented language features in Groovy such as creating classes. The SmartApp can choose to expose web service endpoints to respond to HTTP requests from external application, which is protected by OAuth-based authentication. Note that SmartApps support dynamic method invocation (using the `GString` feature), and thus similar to the reflection feature in some programming languages such as Java, a method can be invoked by providing its name as a string parameter. In later sections, we detail the security problems caused by this dynamic feature and how ContexIoT addresses it.

The cloud backend also runs the *SmartDevices*, which are software wrappers for physical devices in the user's home. A SmartApp and a SmartDevice communicate in two ways (1) The SmartApp invokes operations on the SmartDevices via method call (e.g., to lock the door), (2) The SmartApp subscribes to events that the SmartDevices generates (e.g., smoke detected). The communication between a SmartApp and the functionality of a SmartDevice are controlled by the permission model, which is called the *capability* system of the SmartThings platform.

The current capability model of the SmartThings platform only provides coarse-grained binding between SmartApps and SmartDevices. Capability defines a set of commands and attributes that devices can support, and SmartApps state the capabilities they need. Based on that, users bind SmartDevices to SmartApps at the app installation time. Recent work has uncovered several security problems with the permission/capability system of the SmartThings platform such as *overprivilege* [92]. In this work, we design and implement ContexIoT, a context-based permission system for appified IoT platforms to address these problems.

3.3 Threat Model and Problem Scope

Threat model. In this work, we consider app-level IoT attacks on the appified IoT platforms which attempt to access IoT users' sensitive data or execute privileged functionality. The attacker can launch the attack through either (1) *malware*, in which the malicious logic is embedded at the IoT app install time, or (2) *vulnerable apps*, which contain design or implementation flaws that can be exploited by a co-located malicious IoT app or a remote network attacker to escalate its privilege and cause damages such as unauthorized device control and sensitive data theft. In this work, we assume the platform itself to be trustworthy and uncompromised, thus some recent IoT attacks exploting the unprocted management interfaces of IoT devices to compromise the hardware (e.g., Mirai attack[31]) are not in our scope; Securing the platform by reducing its attack surface is orthogonal to our research (e.g., [63]).

Goal and problem scope. The goal of ContexIoT is to raise the bar for the aforementioned app-level attacks by providing context integrity support. To achieve the goal, ContexIoT aims to enable a user to validate two important properties when a sensitive action is triggered at runtime: (1) *When*: whether the sensitive action is triggered at the user-desired conditions, and (2) *What*: whether the sensitive action matches the user-intended action. Runtime data content validation and protection are also in our scope, since to perform effective access control, the user needs to understand what the data is being accessed or about to be sent.

Since we target app-level attacks, attacks not exploiting app-level vulnerabilities are out of our scope. For example, attacks using stolen external service security tokens due to the weak protection of these external services [92] are considered as a separate problem, and should be taken care of by the provider of each service integrated with SmartThings. Also, the denial-of-service (DoS) behavior of "ignoring the functionality" [144] is not in our scope. For instance, a malicious break-in alert app that claims to notify the user when it detects a break-in may ignore the event instead of sending alerts. In this work, we target

attacks with explicit code-level malicious logic, which can cause more severe damage such as privilege escalation and sensitive data theft compared to DoS.

3.4 Attack Taxonomy

To better understand the security and privacy issues associated with the current appified IoT platforms, we performed an extensive survey of attacks reported on both IoT devices and the smartphone platforms, and studied the feasibility of their migration to the SmartThings platform. For all attacks that are applicable, we constructed misbehaving SmartApps that achieve similar malicious functionality to guide our design and evaluate the effectiveness of our system.

3.4.1 Reported IoT Attacks

Similar to the early stages of any emerging technology, the priority of most vendors are functionalities and faster time-to-market of their products, while security and privacy have not received much attention. The security of IoT platforms are not hypothetical concerns as a number of real attacks have already been reported. For example, IoT devices being compromised to use as bots to launch DDoS attack [162], the misusing of BLE range to confirm physical proximity are leveraged by attacker to unlock your vehicle and door [105, 95]. Table 3.1 lists the reported attack instances we collected from sources including both academic papers and news articles. We categorize them into three classes based on the problem area.

3.4.1.1 Vulnerable Authentication

Authentication plays an important role in the whole lifecycle of IoT devices, and vulnerable authentication is spotted in many critical procedures of IoT devices. For example, a vulnerable device-pairing mechanism may allow attacker to take full control of the device. Due to the lack of displaying functionality in many IoT devices, a management console is

Problem area	Attack description	Platform	Attack vectors	Ref.
Vulnerable authentication	Backdoor pin code injection	SmartThings	Stealing OAuth tokens; Inject command into Web Service SmartApp	[92]
	Get remote shell of device	Telnet-capable IoT devices	Weak/default password; Credential included in the image; Unprotected debugging interface	[136, 162, 39]
	Leaking information,creating seizures	Smart connected LEDs	Unsecured device pairing procedure	[144]
	Impersonate device to steal data	Bonjour-supported IoT devices	Unable to handle name collision in the local network	[65]
Malicious app/firmware	Door lock pin code snooping	SmartThings	Overprivilege due to the SmartApp-SmartDevice coarse-binding	[92]
	Disabling vacation mode	SmartThings	Misusing logic of a benign SmartApp to do event spoofing	[92]
	Fake alarm	SmartThings	Controlling device without gaining appropriate capability	[92]
	Surreptitious surveillance	Sony camera	Installed with malware in the device retailing process	[46]
	Spyware	Barcode scanner	Preloaded with malicious firmware	[14]
Problematic usage scenario	Undesired unlocking	BLE Smart locks	Misusing BLE range to confirm the physical proximity of user	[105]
	BLE relay unlocking	BLE Smart locks	Misusing BLE range to confirm physical proximity of user; BLE Replay attack	[105, 95]
	Lock access revocation / logging evasion	DGC lock	Failing to ensure state consistency between device and server	[105]

Table 3.1: A taxonomy of reported IoT attacks and their applicability to the SmartThings platform

typically provided using protocols such as Telnet, HTTP and SSH, which can suffer from problems such as weak or default password. We find that beside gaining a remote shell on the devices, many attacks can be easily implemented as malicious SmartApps and distributed in the platform. For example, a malicious smart light control app can perform similar malicious activities as described in [144] to use luminance as side channel to inform thieves near the house that the owner is not at home, or creating seizures using strobed lights.

3.4.1.2 Malicious App/Firmware

Even before the emerging of appified IoT platforms, malicious preloaded application or firmware have already been reported [46, 14]. Functionalities of these malicious app/firmware can be easily migrated to SmartThings platform, as it opens a broad range of device capabilities to 3rd party developers. For example, one of our constructed malicious SmartApp show how an attacker can surreptitiously spy on the daily life of house owner if the user installed the malware disguised as normal surveillance camera app. In addition, some attacks that has already been reported as feasible on SmartThings platform [92], such as snooping the door lock pin code, are also included, and used to guide our system design.

3.4.1.3 Problematic Usage Scenario

Another category of IoT attacks exploits the misusing of technology in some IoT specific usage scenarios. For example, using the presence of user's device in the BLE range as indicator of user's presence at the door is considered problematic, since it may undesirably unlock all the doors of the house due to the long range of BLE. We found that such problems can also be reproduced in SmartThings using the capability granted to the SmartApps.

3.4.2 Migrated from The Smartphone Platforms

Security requirements of appified IoT platforms and the smartphone platforms share many similarities, including the definition of access to resources, and privilege separation. We surveyed the mobile malware ecosystem, and categorized them based on the different techniques they used in 4 aspects of their lifecycle below. We discussed the possibility of each malware classes to be appified on SmartThings platforms, and construct real malicious SmartApps for demonstration and evaluation purposes if applicable.

3.4.2.1 Installation.

The most common technique seen in mobile malware samples to distribute themselves is to repackage their malicious app logic into commodity apps that claim normal functionality. This attack venue is clearly applicable to IoT malware. Moreover, we find that the app update procedure, which is reported to have been leveraged by mobile malware to carry out their malicious payload is also vulnerable in the SmartThings platform. SmartThings makes it very convenient for SmartApp developers to deploy their updates, by automatically updating the cloud instances of the SmartApp for all the user. In this mode, the attacker can disguise the malicious logic of their apps by not piggybacking the entire malicious payload into the original app, but slowly introducing it through future updates. In addition, drive-by download can also be easily adopted by attacker to entice users to download the malware app.

3.4.2.2 Activation

Malicious logic can potentially be triggered by various events. We categorize these events into three categories: (1) *Remote command* (e.g., incoming SMS), (2) *User events* (e.g., user click), and (3) *System events*. The trigger-action programming model of IoT provides similar flexibility for attacker to embed their malicious app logic into any of the three types of events. Specifically, some IoT events are very informative and may leak

Stage	Category and descriptions	Ref.	ST?
Installation	**Repackaging**: Malicious logic are enclosed into high-profile apps to trick user to download	[74, 169, 73, 107]	✓
	App update: Malicious payloads are downloaded during the app update process for disguising purpose	[159, 169]	✓
	Drive-by Download: Enticing user to download the "interesting" or "feature-rich" apps	[169]	✓
Activation	**Remote command**: Attacker controlled remote input, e.g., incoming SMS	[169, 96]	✓
	User events: Event triggered by the user, e.g., button click	[96]	✓
	System events: Event generated by the system, e.g., boot complete event	[169, 120]	✓
Adversary technique	**Abusing permission**: malicious app logic abuses the privilege granted to the app	[96, 82, 129]	✓
	Exploiting weakness of general system design: generic system mechanisms such as IPC	[153, 64]	✓
	Exploiting weakness of platform specific features: techniques specific to platform, e.g., native code	[57, 58, 127, 121]	✓
	Exploiting system vulnerability: security flaws and bugs in the system e.g., root exploits	[146, 166, 108, 158, 56]	N/A
	Shadow payload: disguise malicious payload using obfuscation or encryption techniques	[169, 140]	✓
	Side channel: carry out malicious payload using covert channel	[83, 165, 167, 78]	✓
Malicious payload	**Remote control**: Taking control of user's device with C&C servers	[169, 120]	✓
	Spyware: Aiming to gather information from the victims without their knowledge	[96, 82, 129, 167, 123]	✓
	Adware: Downloading and displaying unwanted ads on the user's device	[145, 120, 107]	✓
	Ransomware: Installed covertly to DoS the device and demands a ransom payment to restore it	[115, 108]	✓
	Privilege escalation: Exploiting a bug or design flaw of the system to gain elevated access	[146, 158, 168, 121]	N/A

Table 3.2: A taxonomy of smartphone malware classes and their applicability to the SmartThings platform

```
input "switch", "capability.switch", title: "The switch your camera is
    controlled by"
// subscribe the mode change event
subscribe(location,"mode",handler)
def handler(evt){
//turn switch off if the owner has left
    if(evt.value == "Away"){
        switch.off()
    }
}
```

Figure 3.2: Code snippet of surveillance disabling attack

sensitive data to untrusted apps that don't have essential capability. For instance, the mode change (home/away/night) events are broadcasted system-wide and an malicious app that doesn't have the access to any sensing devices can know when the house owner is leaving by receiving the broadcast, and facilitating potential break-in.

3.4.2.3 Adversary Technique

Two basic principles that guide the design of malware are to (1) carry out the malicious payload as fully as possible under the system constraints to achieve maximum benefit; (2) evade detection to prolong their life-time. Guided by these principles various adversary techniques are used that we categorize into 6 classes shown in Table 3.2. Except exploiting system vulnerability, such as root exploits which is orthogonal to our research, techniques in all other 5 categories can be applied to the appified IoT platforms. For example, the permission mechanism of commodity platforms offers "all or nothing", meaning that once the permission is granted, the privilege can be used for any purpose. This allows malicious app logic to abuse the trusted granted to the declared benign functionality of the same app. Such *overprivilege* are common in SmartThings platform where a AutoLock SmartApp also has the capability to unlock the door anytime [92]. Another interesting evasion technique is to

use IPC between malicious apps to carry out malicious payload. and we demonstrate on SmartThings that even IPC is not supported by the platform, malicious SmartApps with least privilege can collaborate to leak sensitive data such as door lock pin code through the device status as side-channel [92]. In addition, weaknesses in platform specific features can also be leveraged by IoT malware. For example, the `GString` support of the Groovy language enables attacker to modify the control flow of the app at runtime, which can be used to evade all static analysis based malware detection systems.

3.4.2.4 Malicious Payload.

Existing smartphone malware can be largely characterized by their carried payloads. We partition these payloads into five different categories:*remote control*,*spyware*,*adware*,*ransomware*, and *privilege escalation*. Among them, privilege escalation leverages the vulnerabilities of the system, and is out of the scope of our work. Remote control and spyware are two common types of payload on the smartphone platforms and can be easily adopted by IoT malware. Adware is a type of app that downloads and display unwanted ads to the user. There are many channels including push notification and SMS in IoT platforms that can be leveraged by adware to spread ads. Ransomware is an emerging threat to modern systems, and we demonstrate examples showing that IoT malware can also demand ransom payment in situations where the effect caused by the ransomware cannot be easily reverted (e.g., when the user is on vacation).

Among all the 29 categories of attacks shown in Table 3.1 and 3.2, 25 of them are in the scope of our research and are the attacks that our proposed system is designed to defeat. Using these 25 categories, we implemented 25 malicious apps corresponding to each of the category on SmartThings platform, and evaluated our system against them in §3.7. We provide all of malware samples developed in this project on our website [37] to benefit future research. Below, we will provide more detail about three instances of the proof-of-concept attacks we have implemented. We will refer to these three attacks in later sections

```
1 input "lock","capability.battery", title: "The device you want to have its
    battery monitored"
2 // subscribe the battery report from the lock
3 subscribe(lock,"battery",handler)
4 def handler(evt){
5 //transmit battery data to graphing webservice
6     httpPost (url, evt.jsonValue)
7 }
```

Figure 3.3: Code snippet of pin code snooping attack

to show how our design and implementation choices defeat these attacks.

Surveillance disabling attack (Figure 3.2) repackages its malicious payload in an home monitoring app, and abuse the switch control capability granted to this app to turn off the surveillance camera when it detects that the owner has left to facilitate potential break-in. It also leverages the vulnerability in the event system of SmartThings to subscribe on the mode change events without explicitly requiring any capability.

Pin code snooping attack (Figure 3.3) uses a battery monitor SmartApp to disguise its malicious intent at the source code level, and is first proposed in the recent work [92]. The app subscribes on the battery report of the lock, and sends the battery data to remote client for visualization purpose. However, it won't reveal its malicious payload until the victim sets up a new pin code. Due to the *overprivilege* issue of the SmartThings platform, the app subscribing on the battery report can also receive the `codeReport` event when pin code is updated, and user can distinguish the benign and malicious behaviors only based on the runtime value.

Remote control attack (Figure 3.4) leverages the Groovy dynamic method invocation and the asynchronous execution flow to disguise its malicious payload. It pulls the attack server everyday for new malicious command and stores them in the global variables shared by all event handlers. A separated process that is scheduled to run every 5 minutes invokes

```
1  //Subscribe on the sunset event
2  subscribe(location,"sunset",dispatcher)
3  //Schedule the handler to be executed every 5 minutes
4  schedule("0 5 * * * ?", handler)
5  def dispatcher(){
6      httpGet(url){
7  //Query attack server for command and store them in global variables
8          resp->
9              state.method = resp.data['method']
10             state.flag = true
11     }
12 }
13 def handler(){
14 //Execute the command if it's updated
15     if(state.flag == true){
16         "$state.method"()
17         state.flag = false
18     }
19 }
```

Figure 3.4: Code snippet of remote control attack

Name	Description	Definition of context					Decision in context?
		Uid/Gid	UI activity	Control flow	Runtime value	Data flow	
ACG [143]	User-driven access control	✓	✓				✓
AppContext* [161]	Static context-based analysis for malware detection	✓		✓		✓	-
AppFence [106]	Protecting private data from being exfiltrated	✓				✓	
Aurasium [160]	Repackaging app to attach policy enforcement code	✓	✓		✓		✓
CRePE [80]	Enforcing context-based fine-grained policy	✓			✓		
FlaskDroid [70]	Fine-grained MAC on middleware and kernel layer	✓			✓		
SEAndroid [147]	Flexible MAC for Android apps	✓					
SEACAT [82]	Integrating both MAC and DAC in the policy checks	✓	✓		✓		✓
TaintDroid [87]	Dynamic taint tracking and analysis system	✓	✓		✓	✓	✓
TriggerScope* [96]	Static trigger-based analysis for malware detection	✓		✓		✓	-
ContexIoT	Providing contextual integrity to permission granting	✓	-	✓	✓	✓	✓

* These work focus on detecting malicious behavior with static analysis, but not enforcing access control at runtime. However, their methodologies of distinguishing benign and malicious behavior are based on their definitions of context.

Table 3.3: Comparison of the context definitions among related work

the malicious command stored in the global variables using `GString`, which allows attacker to potentially control all of the devices associated with this app.

3.5 ContexIoT Design

Guided by the attack survey and taxonomy, we present the context definition in ContexIoT by identifying a set of information that is essential to distinguish the attack and benign logic in an app at runtime. To better clarify our context definition, we perform a comparison between ContexIoT and previous context-based approaches that aim at detecting malicious app logic or enforcing policies.

3.5.1 Context Definition

We use the term *sink* to refer to all the security sensitive actions of the app in later sections. As shown in Table 3.3, we extract the context definitions from a list of representative related work and categorize them into 5 classes:

UID/GID. For app-level access control mechanisms, the context used to make permission granting decision is the identify of the app, i.e., the UID/GID from the system perspective. Mandatory Access Control (MAC) and Discretionary Access Control (DAC) systems on the smartphone platforms are several examples that use this context definition [70, 147, 82], but they are not able to distinguish attack and benign app logic within the same app.

UI Activity. Runtime access control systems on mobile platforms put user in the context of an app's UI activity to make permission granting decisions [143, 160, 87, 82]. The problem of using UI indicators alone is that it cannot restrict how the app uses the sensitive data. In addition, since the UI is generally not available in the IoT apps due to the design goal of minimum user involvement, it cannot be integrated into the context definition of permission systems on the IoT platforms.

Control flow. The events that trigger the execution of the payload, and the conditional statements (e.g., environmental attributes controlling the execution of payloads) consist of the control flow data in the definition of context. The control flow context of a sink is useful to distinguish attack and benign execution paths. For example, a door `unlock()` action triggered by a remote command is more suspicious than that triggered by entering the correct pin code. However, not all malicious behaviors can be distinguished using control flow context alone, the data floating on the execution paths at runtime is also necessary for making proper permission granting decisions in certain scenarios.

Runtime value. In our attack survey, we find that the same control-flow path can be used either for benign purpose or carrying out attack payload depending on the runtime value of the variables that are related to the security sensitive behavior of the app. As shown in the pin code snooping attack (Figure 3.3), the control flow paths of receiving the battery report and pin code update report are the same, and it depends on the runtime value to distinguish them. However, using runtime value to check contextual integrity causes usability problems since even a tiny change in the data results in different context, and user

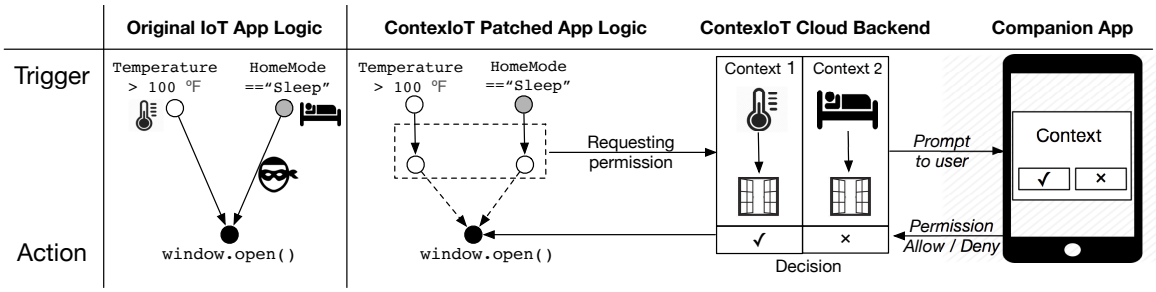


Figure 3.5: ContexIoT overview with a concrete example showing our context-based access control

can be overwhelmed with the large number of decisions to make. Moreover, presenting raw runtime data may not necessarily inform the user about the whether the data is sensitive or not, especially when the malware uses shadow payload technique shown in Table 3.2 to conceal the content.

Data flow. The data dependency information in the data flow context is critical to communicate the context to the user. Integrating it into the context definition mitigates the problems mentioned above by (1) reducing the number of different context by merging the runtime data that come from the same data origin, and (2) tagging the data dependency information to the runtime values to help user make more informed permission granting decision based on the property of data being sent out.

As shown, besides the *UI activity* that is generally not available for IoT apps, ContexIoT integrates all the other context components and thus has the most comprehensive definition of context among related work. Later in §3.5.3, we further use evasion attack discussion to show how this comprehensive definition can help improve the permission system effectiveness.

3.5.2 ContexIoT Approach

We next present ContexIoT, our approach that provide contextual integrity to the permission granting process of IoT apps using the context definition defined in §3.5.1. As shown in Figure 3.5, the general design of ContexIoT consists of two major steps: (1)

At the app installation time, ContexIoT patches the app with security-focused logic to collect essential context and separate the execution flow of the security sensitive behaviors into asynchronous procedures: first request the permission in the current context, and then perform the action when receiving permission granting response. (2) At runtime, the cloud-backed permission management service handles the request from the patched apps and prompts to the user with the context if necessary. Figure 3.5 shows an example in which a malicious home temperature control app is granted with the capability to control the window based on the temperature. However, the attack logic embedded in the app code covertly opens the window when it detects the mode of the home is changed to `Sleep`, which allows the attacker to break in. The ContexIoT patched app puts the open window execution on hold and sends the collected context information to the backend. If previous decision for this context is not found in the backend, the permission service prompts user and takes the permission granting decision that is made in context. The permission service learns the security preference under this context of the user to prevent unnecessary prompts in the future. We introduce how ContexIoT approach collects and uses the context as follows.

Context collection. To overcome the black-box nature of the cloud-backed IoT platform, ContexIoT patches the context collection logic to the app code, allowing the patched apps to gather essential information of their own running context without requiring system access. However, precisely tracking all the control and data flow attributes of the app requires adding the logging logic to almost every instruction in the app code, which may at least double the computation overhead. To address this challenge, ContexIoT takes an hybrid approach combining static analysis and runtime logging to collect essential context efficiently – using static analysis results to reduce the overhead of runtime logging.

The static analysis first identifies all the potential sinks, which are those secure sensitive behaviors in the app code, and constructs an Inter-procedural Control Flow Graph (ICFG) from the program entry points to the sinks. App code that is not on any control-flow path

from the entry points to the sinks does not need to be patched with the runtime logging logic since it won't affect the behavior of the sensitive action. However, some exceptions need to be made for the app logic that may implicitly affect the sinks, which are detailed in §3.6. In addition, some context information that are deterministic statically that doesn't depend on runtime values are precomputed by the static analysis and annotated on the statements of the app code to further reduce runtime computation overhead.

ContexIoT then efficiently patches the app with the context collection logic. The general approach is to maintain an environment variable to store the context information for each application variable that are labeled as related to the program sink by the static analysis. The environment variables are automatically updated during the execution of the app based on the logic implemented by ContexIoT. And when the sensitive execution is triggered at runtime, a context collection function gathers the essential information from the environment of all the variables along the execution path. And the context information is sent to the backend to request permission for executing the sensitive action.

Context usage. Among the different types of information in our context definition, the control flow information, which describes the triggering action of the sensitive action, together with the runtime data should be able to distinguish the context of attack execution path and benign execution path. And the data flow information is used to communicate the context to user to better inform the user the security implications.

As shown in Figure 3.5, the backend permission service maintains an authorized permission-context mapping table for each user. Every time when a ContexIoT patched app attempts to perform a security sensitive action, a permission request containing the context information is sent to the backend. The cloud-based permission service checks whether the context has been previously allowed or denied. If not, it prompts user with a dialog presenting the permission request and the associated context and adds an additional entry to the mapping table to store the user's decision as security preferences. The context structure contains the 4 out of 5 context components described earlier in §3.5.1. Our

Name	Evasion attacks			
	Asynchronous leakage	Control flow abuse	Dynamic code loading	Policy abuse
ACG	×			
AppContext*			×	
AppFence	×			
Aurasium		×		
CRePE	×			
FlaskDroid				×
SEAndroid				×
SEACAT				×
TaintDroid		×		
TriggerScope *			×	

* These work focus on detecting malicious behavior with static analysis using context, and are thus vulnerable to dynamic code loading.

Table 3.4: Evasion attacks on context-based security approaches

definition cannot include the UI Activity class since it is not available in IoT apps. In the implementation, some optimization can be applied to reduce the frequency of prompts by merging some components, which is detailed later in §3.6.

3.5.3 Comparison with Other Context-based Security Approaches

Since our context definition contains the complete inter-procedure control and data flow information, it can distinguish any attack logic in the app code level. To show the effectiveness of this design, we compare our context definition with other context-based security approaches proposed by previous work for smartphone platforms. In Table 3.4, we list a set of evasion attacks that can bypass those systems but can be defeated by ContexIoT. These evasion attacks fall into 4 categories as follows.

Asynchronous leakage. Sensitive data can be leaked to remote attacker stealthily, where the accessing and the transferring of the data are executed asynchronously, using global variables or other sources to share the data between the two procedures. The remote control attack shown in Figure 3.4 is one such example. Access control mechanisms without information flow tracking support [143, 106, 80] can be evaded by malware that abuses

the granted access to resources and covertly leak them to the attacker. ContexIoT defeats such attack by integrating data dependency into the context definition. For the remote control attack, when the malicious payload is executed, ContexIoT presents users with the data dependency information for the sensitive action, which explicitly tells user that the method about to be executed comes from the response received in a separate procedure.

Control flow abuse. Access control systems that enforce policies only at the granularity of sinks without tracking how the sink is triggered [160, 87] is vulnerable to malware that abuses control flow to carry out malicious payload. For instance, a malicious lock manager app is granted with the `unlock()` capability by such sink-based access control systems the first time it attempts to unlock the door when it detects the house owner is back. However it can reuse the same code snippet that has already been permitted to unlock the door upon the remote attacker's request. In the context design of ContexIoT, the inconsistency of the control flows in these two scenarios are then detected, and the user's permission are required separately.

Dynamic code loading. Static analysis based malware detection approaches [96, 161] can be evaded by dynamic code loading. The `GString` support of SmartApps allows malicious payload to be revealed only at runtime. ContexIoT statically detects potential sinks for dynamic code loading, thus prevents malicious logic from evading the access control.

Policy abuse. MAC and DAC based approaches [70, 147, 82] grant the access at the application level based on user or system defined policies. And they are intrinsically vulnerable to malicious app logic that abuses the trust granted to the app itself, which is usually seen in repackaged apps. ContexIoT performs access control on the program path level and can enforce finer-grained policy to distinguish benign and attack logic in the same app.

As shown, our fine-grained context definition at inter-procedure control and data flow levels can successfully defeat these evasion attacks that can bypass other context-based security approach, showing that such a comprehensive definition can greatly improve the access control effectiveness in permission systems.

3.6 Implementation

We build the ContexIoT mechanism on the SmartThings platform, which supports yet the largest number of device types (204) among all the appified IoT platforms [44]. SmartApps are executed in the proprietary Samsung backend and our prototype of ContexIoT automatically patches the app before they are submitted for execution. The patched security logic communicates the context information collected at app runtime to our own backend permission management server when security-sensitive behavior is triggered. We detail the key components of the app patching mechanism of ContexIoT and the end-to-end implementation.

3.6.1 SmartApp Patching Implementation

Recall that our context definition in §3.5, which contains control and data flow attributes and also runtime values. ContexIoT enables the patched SmartApp to collect these context information at runtime by adding the logging logic to the instruction set of the original app. To reduce the runtime computation overhead of maintaining the context for the whole program, ContexIoT also employs a static analysis approach to (1) Identify a subset of the app code that require runtime logging to track the sensitive execution, and (2) Precompute some context information that is deterministic statically. Based on the annotations done by the static analysis, ContexIoT efficiently adds instructions to log the context attributes only for the subset of app code that is related to the sensitive behaviors of the untrusted app.

We define the *sinks* of SmartApps as the security-sensitive behaviors of the app, which contain both capability-protected APIs that are used to control or actuate the device, and other security-critical APIs such as `sendSMS()` and `setLocationMode()`. As of July 2016, 83 device-control APIs protected by 67 capabilities are supported by SmartThings, and will be recognized as sinks in our analysis. In addition, we also consider a set of SmartApp APIs that can be potentially used by attacker to carry out malicious payload. For example, malware can use the `setLocationMode()` to disarm the house by changing the mode to

```
//Define Web API to get permission response
path("/response/:data"){ action[POST: "onResponse"] }
state.actionQueue = []
subscribe (location, "mode", handler)
def handler(evt){
  if(evt.value == "Away"){
//Pseudo function that collects context
    def context = collect_context()
//Enqueue the sensitive app logic
    state.actionQueue << [device:"switch",command:"on()"]
    def id = state.actionQueue.length
//Request permission with the context
    httpPost(url,"['$id,$context]")
}}
def onResponse(){
  def id = params.data.id
  def permission = params.data.response
  if(permission == "Allow"){
//Dequeue the sensitive execution
    action = state.actionQueue[id-1]
//Execute the sink using dynamic code loading
    if('switch'==$action['device']){
      switch."$action['command']"()
    }
}}
```

Figure 3.6: Code snippet showing how the surveillance disabling attack app is patched with the ContexIoT secure logic

"*Home*", use `httpPost` to leak sensitive data, and use `sendNotificationToContacts()` to send phishing messages to the victim's contacts. We therefore collected 36 such APIs and added them to the sink API set.

3.6.1.1 Static Analysis

To model the lifecycle of the SmartApp and computes the minimum set of app code that can potentially affect the behavior of the sink, ContexIoT builds an ICFG for the SmartApp. The ICFG is constructed using the AST transformation support of Groovy language [16], which allows the static analysis to be performed directly on the Abstract Syntax Tree (AST) generated during the compiling process. Specifically, In the programming model of SmartApp, the app is not continuously running, app logic is embedded in different event handlers that are triggered by the events they have subscribed on. We adapt our design to the trigger-action based programming model of SmartApps, and models all the program entry points that can potentially be triggered by runtime events. In general, ContexIoT doesn't patch the app code that are not in the ICFG from the program entry points to the sinks. However, one exception is that it will patch all the statements that modify the value of global variables, which are shared among executions, since they will also determine the behavior of the sinks. The remote control attack shown in Figure 3.4 is one example.

The static analysis of ContexIoT further reduces the runtime overhead by precomputing the intra-procedural control-flow context for program statements, which doesn't contain dynamic method invocation (`GString`) in their control-flow paths. Similar to reflection, the dynamic method invocation support of Groovy can modify the control-flow at runtime. The dynamic features will be handled using runtime logging, however, except that, the intra-procedure control-flow information for all the other statements are deterministic statically, and ContexIoT annotates these statements with the precomputed context. Based on the call trace collected at runtime, the complete control-flow context for a certain sink is obtained by composing the annotated intra-procedural context of all the statements along the method

invocation chain.

3.6.1.2 Runtime Logging

Excluding the intra-procedure control-flow context that is already been computed and annotated statically, there remain four major types of information that are required to be collected at runtime, in order to complete the context definition. (1) *Method invocation trace* is logged by adding a variable for each method in the app to keep track of its calling function. Every method call expression will set the variable of the callee function with the calling function's signature. Once a sink is triggered, the call trace can be extracted by tracing back the method signatures stored in these variables. (2) *Dynamic method invocation* is captured by using a variable to track the value of each `GString`, which can only be determined at runtime. For example in the remote control attack shown in Figure 3.4, when the sensitive execution in the `handler` is triggered through dynamic method invocation, the patched SmartApp will gather the device and the method name and put them into the context. (3) *Runtime data* can be directly obtained from the variables without adding new instructions to track them. When a sink is reached at runtime, the context collection logic gathers the current value of all the variables that the sink statement is control-dependent on, and use them as the runtime data in our context definition. (4) *Data dependency* is critical to communicate the context with the user as described in §3.5, and we design and implement a dynamic taint analysis framework to track the data dependency information of sink-related variables.

The dynamic taint analysis of ContexIoT maintains a *taint environment* for each variable in the subset of app code output by the static analysis. Each program variable v is associated with a taint environment $\gamma : v \rightarrow T$, where T is a set of taint values $\{t_i | i = 1, ..., k\}$. Each taint value t_i is associated with a variable v_i, meaning that v is data dependent on variable v_i. In our design, variables in the taint environment $gamma$ include local, global, and function return variables, and are maintained as `JSON` objects in our implementation.

Specifically, in the SmartThings programming language, the only global variable is the `state` object, which allows developer to store data into different fields of the object and shares the data across executions. And our taint logic is designed to be field-sensitive to precisely track the data dependency relationship of all the global variables in different fields of `state`.

The taint propagation of ContexIoT follows the generic approach [76] and handles some Groovy-specific operations such as the array insertion operation ($<<$), `closure`, and also the library functions of SmartThings. We manually summarized all the 85 SmartThings APIs available as of June 2016 in a file, which specifies how the taint values are propagated through the function variables to the return value. We also considered the side effect when modeling these library functions, which is the potential impact the functions have on the global variables. For example, once the `changeLocationMode()` function sets the global variable $Mode$, it will affect all the variables that depends on it. Our analysis handles such case by updating the taint environments for the corresponding variables when such function calls are executed.

In addition, ContexIoT also considers implicit flows, where the taint value is in the conditional statement that the sink is control dependent on. The implicit flow helps capture data dependency that is not directly propagated by assignments, and enables the detection of information leakage through side-channel. For instance, a malicious app we have constructed uses the light luminance as side-channel to send sensitive information such as the home occupancy and the lock status to attacker nearby, which will not be detected by explicit data flow analysis, but can be captured by implicit flow. We label each taint value with 2 boolean values E and I in its taint environment, for taint value coming from explicit flows ($E = true$) or implicit flows ($I = true$). When merging two taint values with different labels E_1,I_1 and E_2,I_2, the merged taint value's label is ($E_1||E_2$) and ($I_1||I_2$).

3.6.1.3 Secure logic patching

ContexIoT separates the execution flow of the sink into two asynchronous procedures: first requests the permission with the collected context, and then resumes the execution upon permission granting response. Figure 3.6 shows the code snippet of the surveillance disabling attack logic being patched by the ContexIoT. The original sink (`switch.off()`) is modified to the secure logic of enqueuing the action for future reference, and sending the collected context to permission server. ContexIoT patches the app with a Web API interface to receive the permission response in an separate process to overcome the restrictions of SmartThings on the execution time of each code block. On receiving the response from the server, the `onResponse` handler retrieves the information of the sink from the queue and execute it, if the server allows the execution in this context. The performance overhead of adding such secure logic is evaluated in §3.7.

3.6.2 End-to-End Implementation.

We set up the ContexIoT cloud backend service on a cloud instance in Google App Engine, which stores the previous user permission granting decisions. If no previous decision is found, it prompts the user using Google Cloud Messaging through the ContexIoT companion app to display the context and learn the user's decision. The basic approach for the permission service to distinguish two context is to compare the values of all the context attributes. However in our implementation, differences in the runtime value may be ignored under certain circumstances. The data floating in between SmartDevices and SmartApps are usually in the format of key-value pair (KVP), and in our threat model, we trust the SmartDevices to provide the authentic KVP, where the key reflects the real property of the data. Thus the runtime value are compared by their keys in our implementation, and we use the pin code snooping attack (Figure 3.3) to show that this design choice reduces the prompt frequency, while maintains the effectiveness. The malicious battery app subscribes on the device reports, and if the KVP of two battery reports are presented in

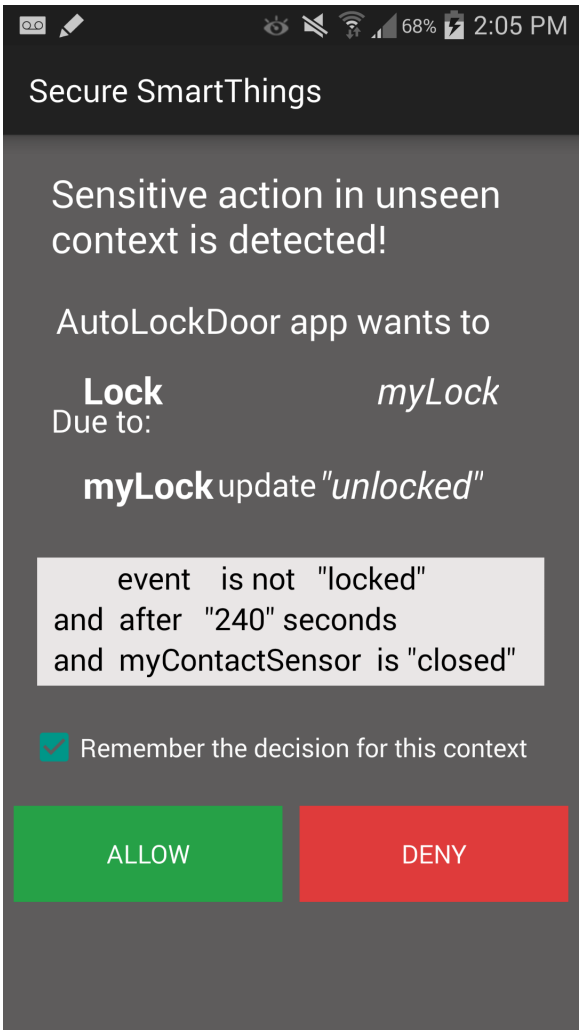
the context as ["$battery\_level$" : 99] and [$battery\_level$" : 95], they will be considered as the same; However, if a pin code update report is sent out, the KVP in the context, which looks like ["$code$" : 9998], will be distinguished from previous paths, and prompted to user separately.

Figure 3.7 shows screenshots of the presented context information for a malware adapted from a real SmartApp called AutoLockDoor [11]. The legitimate functionality of this app is to automatically lock the door after a certain period of time, which is set by the user. In addition it checks the contact sensor of the door before issuing the `lock()` command to ensure that the door can be properly closed. Figure 3.7a shows the permission dialog for this legitimate execution path, which is consistent with the app description. However, this malware also includes a backdoor, which allows the attacker to unlock the door remotely via network commands when the user is away. As shown in Figure 3.7b, since this malicious logic is distinguishable in the control and data flow level, this backdoor logic is revealed clearly in the displayed context information.

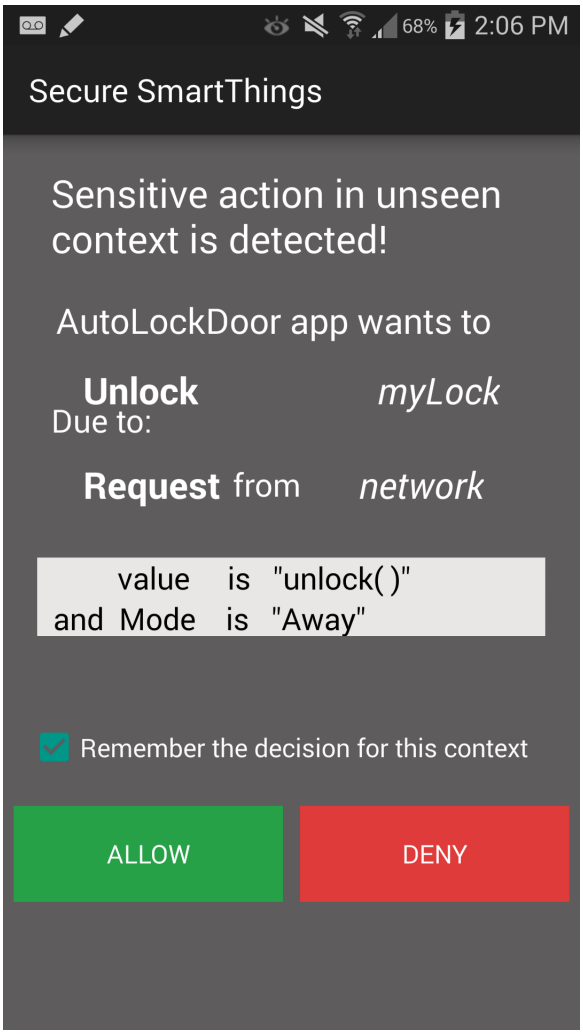
It is important to note that this is only a proof-of-concept context presentation which dumps everything in the context structure to the dialog. Since our context definition contains the complete inter-procedure control and data flow information, future IoT platforms which adopt the ContexIoT approach can flexibly tune the context granularity, e.g., by shortening the length of recorded control flow or merging current context components, and also design better context presentation to meet their usability requirements.

3.7 Evaluation

In this section, we evaluate our prototype implementation of ContexIoT in (1) Effectiveness of secure logic patching; (2) Permission request frequency, which is important for the effectiveness of runtime permission system in practice [157, 57]; (3) Runtime performance overhead of the additional patching logic.



(a) Legitimate logic: Automatically lock the door after a specified period of time



(b) Backdoor logic: Unlock the door when a remote command is received from the network

Figure 3.7: Screenshots showing the benign and attack context in the malicious AutoLock-Door app

3.7.1 Effectiveness of Secure Logic Patching

To evaluate the secure logic patching mechanism, we use ContexIoT to patch the 25 SmartApps we constructed, each representing a unique class of malware or an vulnerable app based on our IoT attack taxonomy in §3.4. The SmartThings IDE provides a simulator that can model the behavior of native SmartThings devices without requiring a physical device [45]. Unfortunately, 3 of the attacks in our taxonomy involve 3rd party devices, for example a camera with advanced features used in the surreptitious surveillance attack [34], and thus cannot be dynamically tested. Thus, we test our system against 22 attacks in the runtime, and only manually examine the patched SmartApp code for the remaining 3 apps.

For effectiveness evaluation, we first check whether all the potential sinks in these SmartApps are patched with the secure logic, and find that ContexIoT accurately identifies and patches all 72 potential sinks including dangerous usage of `GString`. Next, we evaluate whether the attack execution paths in these SmartApps can be distinguished from the remaining benign paths in the runtime. By triggering all the program paths of the 22 attack execution paths, we confirm that all of them can be successfully recognized without any ambiguity, and the other 3 attacks can also be identified based on the statically computed intra-procedural context information. Overall, these results show that our secure logic patching can accurately identify sinks and logging essential context to distinguish attacks execution paths.

3.7.2 Permission Request Frequency

Experimental setup. To measure permission request frequency, we dynamically trigger the execution paths in a set of SmartApps using the SmartThings IDE simulator. In the experiment, we use 283 SmartApps out of the 502 commodity SmartApps we collected since the rest of them can not be simulated due to limited physical device support in the current simulator. To automate the test, we leverage the trigger-action programming model of SmartApp, i.e., app logics are all triggered by external events, to generate inputs.

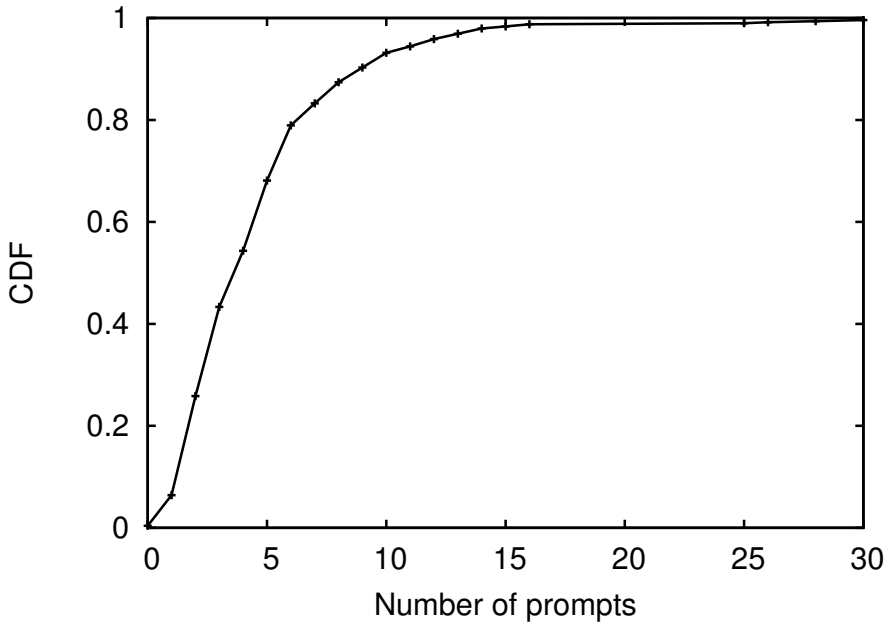


Figure 3.8: CDF of the estimated life-time permission request prompts for 283 commodity SmartApps patched by ContexIoT

Leveraging the events generation support in the SmartThings IDE, we build an automatic SmartApp dynamic testing framework using web automation technique that can generate input to SmartApps and efficiently trigger different event handling logic.

Using our dynamic testing framework, we measure the life-time permission request number for each SmartApp by triggering all possible execution paths in the app. This is an upper bound estimation for a home user in practice, since typically a user only use a subset of all features in an app. For each SmartApp, we use the fuzz testing approach to randomly generate all types of external events in different triggering order to ensure good code coverage. Once a sink is triggered, we log the permission requests, and automatically grants permission to avoid double counting. The test stops only if no prompts are generated after 50 consecutive random events. As shown in Figure 3.8, the average life-time permission request number among the 283 SmartApps are only 3.5, which is far below the threshold that is considered to risk user habituation or annoyance [160, 157].

3.7.3 Runtime Performance

In this section, we measure the performance overhead on the event handling latency introduced by the secure logic patching in ContexIoT. The end-to-end latency for an event execution can be broken down into 2 parts: (1) *computation* latency, which is the time taken in executing the app code, (2) *sink execution* latency, which is the time taken in communication between the SmartThings cloud backend and the physical device. In ContexIoT, the secure logic patching adds latency to the *computation* part since additional instructions are added to the SmartApp used for tracking the control and data flows. At the same time, it also adds *permission request* latency, which is taken in communicating with the ContexIoT cloud backend for permission granting. In this evaluation, we only measure the latency added by the ContexIoT system itself, and thus the decision making time taken for a user is not considered.

Using our dynamic testing framework, we inject events to trigger all the 916 event handling logic in the 283 SmartApps, and the measurement results are shown in Figure 3.9. Overall, we observe 67.1 ms (26.7%) additional latency on average when running those patched SmartApps on virtual devices. In addition, we find that the end-to-end latency is dominated by the sink execution latency, which is at least one magnitude higher than the additional latency from ContexIoT secure logic patching. Thus, we believe that the performance overhead from ContexIoT is negligible in real world scenarios.

Besides testing on virtual devices, we also evaluate the performance overhead for two events using physical devices: (1) Locking a Schlage Z-Wave lock [42] using a commodity lock manager SmartApp, and (2) Sending SMS to the user's phone from a SMS alert SmartApp once it detects motion sensor event. We trigger both events 50 times, and as also shown in Figure 3.9, the patched logic added only 9.6% and 4.5% delay on average. The breakdown of the end-to-end latency shows that compared with running on the virtual device, the time it takes to execute the command on physical device is significantly longer, which is likely due to the latency introduced by the wireless communication be-

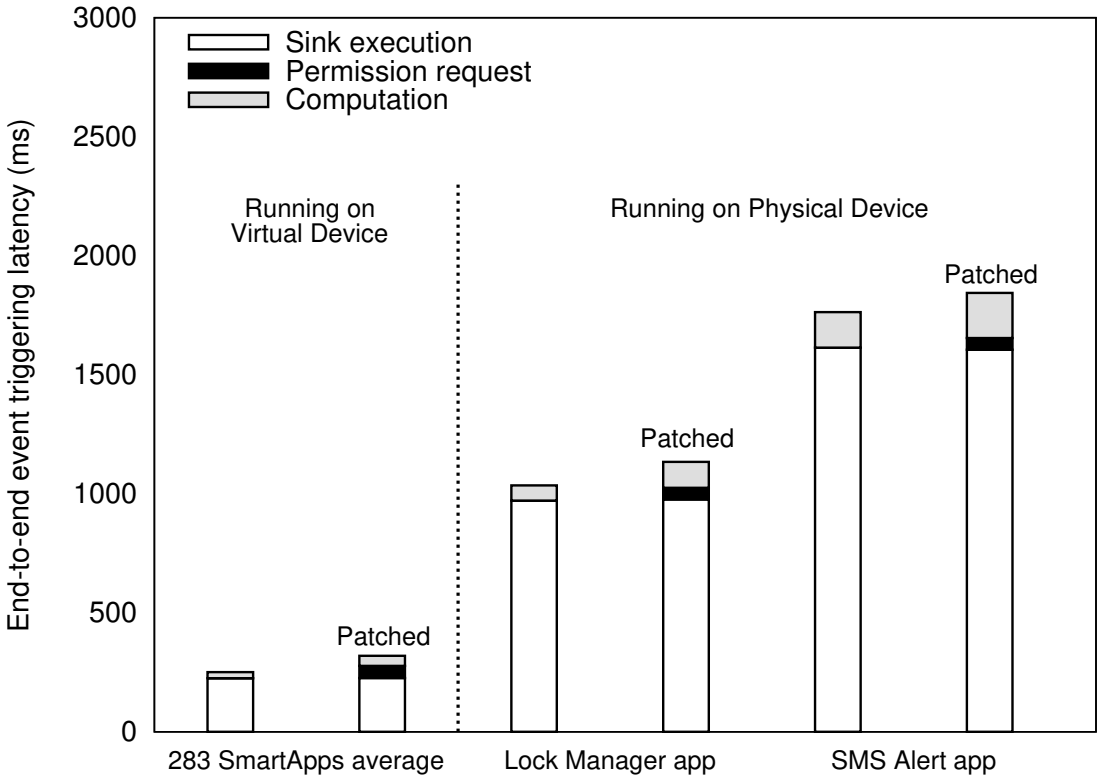


Figure 3.9: Breakdown of the end-to-end event trigger latency with and without ContexIoT patching on virtual and physical devices

tween hub and the device. Thus, the overhead of ContexIoT introduced in the computation and permission request procedures becomes negligible in the physical device settings.

3.8 Usage Discussion

Acar *et al.* suggests to take both users and developers out of the loop as a potential solution to the permission comprehension problem [57]. Their proposed approach for achieving this is to enable the automatic generation of security policies. We believe that the rich context definition and flexible design of ContexIoT benefits innovations in this direction. For example, one potential approach can be to provide recommended context-based security settings to users for different apps, and the recommended settings come from the security preferences that are learned by ContexIoT from a group of expert users using the

ContexIoT-patched apps.

Another possible approach that takes human out of the loop is automatic generation of policy based on the app logic and its interactions with user. For example, the SmartApp has a setup procedure during the app installation, which requires user to set some parameters that will guide the automation of the SmartApp (e.g., automatically locks the door when it has been opened for 2 minutes). Using the data-dependency tracking support of ContexIoT, the security logic can monitor how the app uses the user input. If it detects that the events corresponding to the unmodified user input triggers the user's desired action at the runtime, the execution can be automatically allowed since it conforms with the user specified routine. Natural Language Processing (NLP) technique may be required to infer the user desired action. We leave it as future work to explore these extended usages of the ContexIoT.

3.9 Conclusion

We design and prototype ContexIoT, a context-based permission system for appified IoT platforms, which can support identification of fine-grained context defined at inter-procedure control and data flow levels, and runtime prompts with rich context information to help users perform effective access control. By comparing our context definition with those in previous context-based permission systems, we shows that our definition is more comprehensive and can defeat attacks that can evade these previous designs. Based on the extensive survey of existing and potential attacks on the appified IoT platform, we demonstrate that ContexIoT can effectively distinguish all attack context in the tested apps. Dynamic testing on 283 commodity SmartApps shows that ContexIoT introduces negligible performance overhead and has a low prompt frequency which is far below the threshold that is considered to risk user habituation or annoyance.

CHAPTER IV

Providing Efficient Dynamic Testing Support to Self-driving Functionalities

4.1 Introduction

The transportation ecosystem is on the verge of its most significant transformation in more than a century. For the first time since Karl Benz took his first drive in 1886, connectivity, automation, and services are being added to the automotive, which have the potential to virtually eliminate automobile crashes, fatalities, and injuries. However, before that goal can be achieved, Autonomous Vehicle (AV) manufacturers must ensure that the possibility of malicious actors interfering the vehicle system be minimized, and the chances and impact for any interruptions of the system or self-driving functionalities be mitigated. This actually poses several new challenges in the quality assurance of AV code base.

First, microprocessors, software, and sensors have been the key technologies enabling the automotive industry to meet increasingly stringent economy and safety requirements. Today's most common vehicles have nearly 100 discrete electronic control units (ECUs) and in some cases more than 100 million lines of code, a significant portion of which makes up the advanced driver assist system (ADAS). Moving towards the AV era, these self-driving related functionalities are becoming increasingly sophisticated and eventually morph into full-blown level 4 or even level 5 autonomous driving system as shown in Fig-

Level	Name	Execution of steering and acceleration/ deceleration	Monitoring of driving environment	Fallback performance of dynamic driving task	System capability (driving modes)
Human driver monitors the driving environment					
0	No Automation	Human	Human	Human	n/s
1	Driver Assistance	Human system	Human	Human	Some modes
2	Partial Automation	**System**	Human	Human	Some modes
Automated driving system monitors the driving environment					
3	Conditional Automation	System	**System**	Human	Some modes
4	High Automation	System	System	**System**	Some modes
5	Full Automation	System	System	System	All modes

Figure 4.1: SAE 6 levels of Autonomous Vehicles

ure 4.1. To ensure all these functions' execute as expected in the complex in-vehicle system under numerous roadside conditions, requires huge efforts for software quality assurance, which has been considered challenging especially in large systems such as the automotive [69]. Moreover, to reduce the development cost, automakers of traditional vehicle usually outsource the development of peripheral functionalities, such as the lock system, GPS modules, etc., and they adopted the Controller Area Network (CAN) bus design shown in Figure 4.2 to connect different subsystems. The modern AV platforms, which are usually built based on traditional vehicles [38, 12, 36, 55], inherit this CAN bus design, and provide software abstraction for the physical CAN bus (Figure 4.3).Self-driving functionalities and peripheral subsystems are all connected through this virtual CAN bus, which makes it an hybrid in-vehicle system where third party code such as libraries, firmwares, outsourced components are all connected and communicated using the IPC provided by the platform. It is still a challenging problem how to ensure the security and safety of the AV platform.

The discovery of software vulnerabilities and bugs is a classic yet challenging problem. Due to the inability of program to identify non-trivial properties of another program, the generic problem of finding software problems is undecidable [142]. In particular, securing

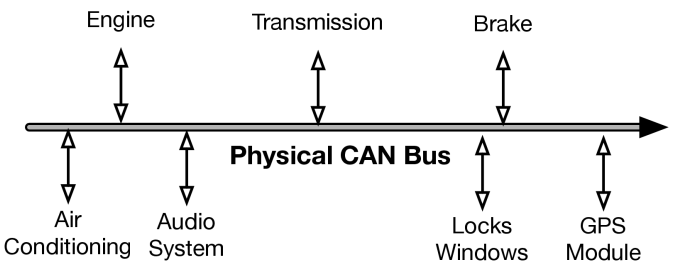


Figure 4.2: Traditional physical CAN bus vehicle platform

large software system, such as an operating system kernel, a self-driving platform, resembles a daunting task, as a single flaw may undermine the security or safety of the entire system. As a result of this situation, security research has initially focused on statically finding specific types of vulnerabilities, such as flaws induced by insecure library functions [19], buffer overflow [118], integer overflows [155], or insufficient validation of input data [112]. However, these static program analysis approaches requires a precise definition of the problem, and they are only as good as the rules they are using to scan with, so that dynamic analysis becomes a necessary approach to uncover unknown programming errors that can potentially cause security and safety problems at the AV's runtime. "Fuzzing" is a practical dynamic analysis approach that works well especially on large code bases where concolic analysis are not applicable due to path explosions [68]. A "fuzzer" monitors the native execution of an application to identify flaws. When flaws are detected, these systems can provide the actionable inputs to trigger them. Although the automakers may adopt general results and existing fuzzing tools from the software engineering body of knowledge gained in traditional software testing domain, the specific constraints and domain specific requirements in the automotive industry ask for specialized solution.

For example, considering a controller module on the AV platform that processes the planned trajectory messages from the path planning module as shown in Figure 4.4. The controller parses the message field by field and calculate the control command to be published onto the CAN bus based on the planned trajectory. However, in this case, the checks performed on various message fields are quite different from the fuzzing perspective. For

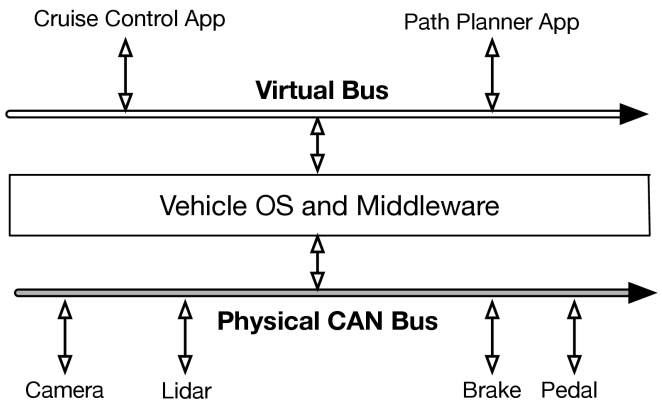


Figure 4.3: Virtual CAN bus design of emerging autonomous vehicle

```
1  header {
2    timestamp_sec: 1497125269.42
3    module_name: "planning"
4    sequence_num: 429
5  }
6  total_path_length: 33.2050000763
7  total_path_time: 10.0006000996
8  estop {
9    is_estop: false
10 }
11 trajectory_point {
12   x: 586392.114303
13   y: 4140672.96946
14   z: -29.2976386519
15   theta: 2.88912842926
16   s: 0.0
17 }
```

Figure 4.4: Example message received by controller module of Apollo AV platform

the comparison of the `module_name` field, a fuzzer randomly mutating input would have a very small chance of sending correct input. In addition, to explore deeper logic of the controller module, the `sequence_num` filed needs to be constructed carefully to bypass the checks, while the coordinates of trajectory points are also subject to certain real world constraints, which also makes random mutations impractical. A domain specific mutation strategy that are aware of all these constraints would be well-suited for recovering the correct field name and valid trajectory coordinates in order to explore deeper logic.

Guided by this intuition, we propose a dynamic testing approach for self-driving functionalities, called AutoFuzzer, that is an enhanced fuzzing tool combining a mutation engine that is specific to the AV domain to identify deep bugs or logical errors in the self-

driving code bases. Combining the domain specific input-mutation fuzzer and a bridge app that connects the fuzzer to the distributed in-vehicle system, AutoFuzzer is able to test complex self-driving functionalities that incorporate multiple modules, to discover systemic issues lying in the autonomous driving logic beyond shallow bugs. We will describe the design and implementation of AutoFuzzer, and evaluate its performance on the Baidu Apollo [12] platform, which is yet the most mature AV platform that are publicly available.

AutoFuzzer is not the first work to improve the fuzzing practice by combining different types of analysis or domain knowledge [103, 131, 148]. However, we show that guided with domain specific knowledge, the performance of the input-mutation fuzzer can be greatly improved to discover bugs such as crashes and hangs more efficiently. Furthermore, we design our mutation engine to be extensible, allowing more vehicular expertise to be translated to new domain-specific atomic mutations to further improve the fuzzing practice for self-driving functionalities. In summary, this work makes the following contributions:

- We propose an improved fuzzing approach to improve the effectiveness of testing self-driving code bases by leveraging domain specific atomic mutations.
- We build the tool, AutoFuzzer on Baidu Apollo autonomous vehicle platform to demonstrate our approach
- We demonstrate the improvement of AutoFuzzer by comparing the number of unique crashs and hangs produced by AutoFuzzer on the Apollo code base with existing fuzzing approaches.

4.2 Background

This chapter provides sufficient background of autonomous vehicle architecture and how the self-driving functionalities are developed.

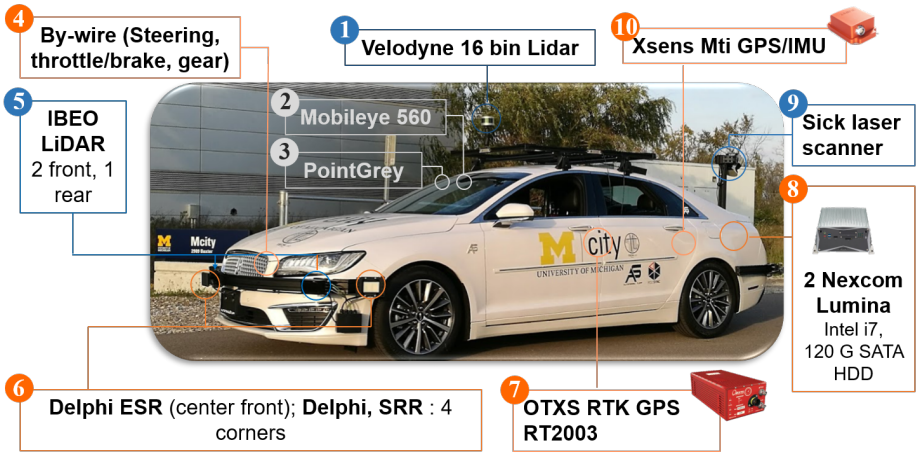


Figure 4.5: The open-access automated vehicle of at the University of Michigan

4.2.1 Autonomous Vehicle Platform Design

To identify key challenges and issues for the AV platform, we compare aspects of the traditional vehicle platform with the modern AV platform design. As can be seen in Fig. 4.2, from the adoption of the Controller Area Network (CAN) bus in the 1980s, modern vehicles now have over 100 electronic control units (ECUs) for various subsystems [59], including critical control systems and also peripheral infotainment systems all communicate using the CAN bus. As automakers usually outsource the development of these peripheral functionalities to reduce development costs, security and safety problems arise,as software developed by a third party with access to CAN bus, can potentially be exploited to tamper the safety of the vehicle. For example, the Jeep hack in 2016 that resulted in a massive recall was due to the vulnerability in the 4G/LTE module [24].

The emerging AV platform shown in Fig. 4.3 provides software abstraction for the physical CAN bus. Self-driving functionalities are developed as apps, and their interactions with the hardware actuators are proxied by a vehicle operating system that also acts as the middleware. The software based AV platform has created an ecosystem where multiple functionalities can work together to provide greater intelligence and convenience. Some organizations have already begun to roll out the autonomous vehicle with open development support. For example, in the industry, vehicle middleware platforms that open

massive vehicle functionalities, including steering wheel and brake to the developers have been built (e.g., Ford OpenXC [36], PolySync [38]). While in the academia, University of Michigan (U-M) starts to offer open-access to their testing AV equipped with sensors, including lidar, radar, and cameras, so that researchers can rapidly test their self-driving or connected-vehicle technologies [49] (Fig. 4.5). Since these emerging AV platform opens full-fledged self-driving functionalities to the third party developers, it could potentially introduce greater safety risk. For example, flaws in the proportional control algorithms of a cruise control app may put the vehicle in a situation where a collision becomes inevitable. Recent accident records [27] of self-driving cars suggest that deficiencies in the AV software are inevitable due to the complexity of the physical environment, thus rendering the vulnerable apps a persistent threat to the AV industry. Apps may also be developed for malicious purposes to tamper with the user's safety with embedded malicious logic [13].

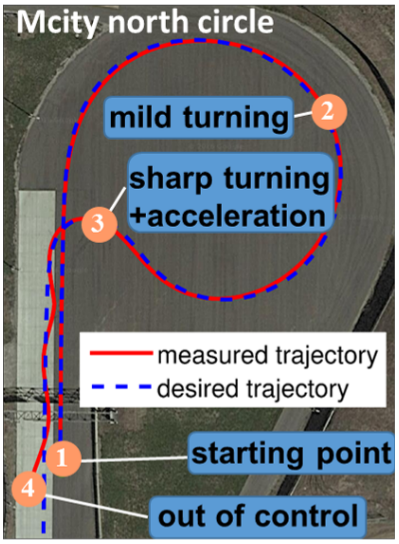
4.2.2 A Motivating Example

Figure 4.6 is the code snippet of a working self-driving app developed by U-M researcher that implements the path following functionality for the AV and enables the vehicle to drive following a given set of coordinates. We use it as a running example to show that although static app vetting is effective for checking the logic with invariant principles, some potential risks may not be revealed, however, until the vehicle encounters certain physical roadside conditions.

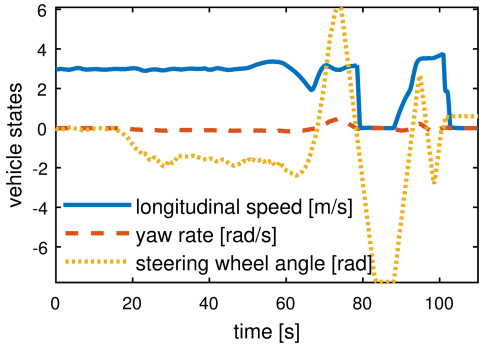
The vehicle status of running the app on the U-M autonomous vehicle in the MCity testbed [29] is plotted in Figure 4.7. The automated vehicle successfully followed the trajectory when the acceleration and curvature were small (point 2), but lost control when the desired yaw rate was high (point 3). The vehicle completely lost stability at point 4 where the driver needed to take over to avoid a collision. Dynamic analysis such as fuzz testing bridges the gap by testing the program ideally in all the scenarios for which it is designed to detect any potential violation of security and safety principles that need to be

```
class PathFollowingNode
{
  //App is a node on Virtual CAN bus
  List<Position> map;
  public PathFollowingNode(){
    //Subscribe on the vehicle report
      registerListener( VEHICLE );
    //Load the trajectory data
      map = loadMap("map.dat");
  }
  //When vehicle report message is received
  void messageEvent (Message msg)
  {
    float _distErr;
    float _headingErr;

    for (long i=startNum; i<map.length(); i++)
    { //Calculate distance error and heading error referring to the path
      _distErr = sqrt(pow(msg.position.x-map[i].x,2)+
        pow(msg.position.y-map[i].y, 2));
      _headingErr = abs(msg.position.heading-map[i].heading);

      Message message = new Message();
      if(_distErr<MAX_DIST_ERR && _headingErr<MAX_HEADING_ERR){
      //If vehicle in path, continue with the adjusted steering angle
        float _angel = CalcSteeringAngel(msg);
        message.setSteeringAngel(_angel);
      //Publish message on virtual CAN
        message.publish();
      }
      else{
      //If vehicle not in path, perform stop with the maximum throttle gain
        message.setThrottleCommand(100);
        message.publish()
        break;
      }
    }
  }
}
```

Figure 4.6: Code snippet of a working path following app



(a) Desired and measured trajectories



(b) vehicles states (longitudinal speed, yaw rate, steering angle)

Figure 4.7: Example of improper dynamic controller that led to out-of-control failure

upheld under any circumstances.

4.3 Related Work

AutoFuzzer is a guided whitebox fuzzer, which builds on top of state-of-the-art fuzzing techniques, adding domain specific mutations to achieve effective bug excavation. As some other fuzzing tools also combine multiple techniques, we use this section to distinguish AutoFuzzer from other solution which draw on related techniques.

4.3.1 Guided Fuzzing

Fuzzing was originally introduced as one of several tools to test UNIX utilities [124]. Since then it has been extensively used for black-box security testing of applications. However, fuzzing suffers from a lack of guidance – new inputs are generated based on random mutations of prior inputs, with no control over which paths in the applications should be targeted.

The concept of guided fuzzing arose to better direct fuzzers toward specific classes of

vulnerabilities. For example, many studies have attempted to improve fuzzing by selectively choosing optimal test cases, honing in on interesting regions of code contained in the target program [103, 131]. Specifically Dowser uses static analysis to first identify regions of code that are likely to lead to a vulnerability involving a buffer overflow. To analyze this code, Dowser [103] applies taint-tracking to available test cases to determine which input bytes are processed by these code regions and symbolically explores the region of code with only these bytes being symbolic. Unlike Dower that targets on buffer overflow vulnerabilities, AutoFuzzer targets on any crashes or hangs, which are considered safety-critical on the AV platform, and can potentially lead to attacks.

In another attempt to improve the state of fuzzing, Flayer [84] allows an auditor to skip complex checks in the target application at-will. This allows the auditor to fuzzy logic deeper within the application without crafting inputs which conform to the format required by the target, at the cost of time spent investigating the validity of crashing inputs found. Similarly, Taintscope uses a checksum detection algorithm to remove checksum code from applications, effectively "patching out" branch predicates which are difficult to satisfy with a mutational approach [154]. This enables the fuzzer to handle specific classes of difficult constraints. Both these approaches, however, either require a substantial amount of human guidance in Flayer's case, or manual efforts to determine false positives during crash triaging. AutoFuzzer address the challenge of bypassing non numerical checks by adding vehicle specific atomic operations to the mutation approaches, so that inputs generated by the mutation engine has a very high chance of passing the check. Compared with Flayer's approach, AutoFuzzer theoretically produces less false positives by constructing realistic input to satisfy first few levels of constraints.

Another approach is hybrid fuzz testing, in which limited symbolic exploration is utilized to find "frontier nodes" [137]. Fuzzing is then employed to execute the program with random inputs, which are pre-constrained to follow the paths leading to a frontier node. Along this direction, several implementation-agnostic fuzzers have been proposed recently

that leverage either model-guided approach [110, 126] or the dependency among the bit positions of an input [72] to improve the efficiency of fuzzing. Specifically, the model based attack generation in TCPWN [110] leverage the domain knowldge about the TCP state machine to inject the attack at the right time during execution, while SemFuzz [72] augments guided fuzzing by tunning the mutation ratio from a given program-seed pair to an optimal value based on the probability of finding crashes. These methods are proved to be very useful for ensuring that the fuzzed inputs explore paths faster in the execution of the binary, but they are generally limit to certain problem scope since the models in TCPWN and the seeds generation in SemFuzz all require domain knowledge. These approaches are very similar to AutoFuzzer in terms of combing domain-specific mutaions, except that AutoFuzzer leverages vehicular domain knowledge to guide the mutation, and is thus more effective specifically in the testing of AV functionalities.

4.3.2 Whitebox Fuzzing

Other systems attempt to blend fuzzing with symbolic execution to gain maximal code coverage [71, 98, 99]. These approaches tend to augment fuzzing by symbolically executing input produced by a fuzzing engine, collecting symbolic constraints placed on that input, and negating these constraints to generate inputs that will take other paths. While these systems are powerful, they suffer from a fundamental problem: if a conditional branch depends on symbolic values, it is often possible to satisfy both the taken and non-taken condition. Thus the state has to fork and both paths must be explored. This quickly leads to the well known path explosion problems, which is the primary inhibitor of these white-box fuzzing when used on large code bases.

AutoFuzzer on the other hand, builds itself on a popular off-the-shelf fuzzer, American Fuzzy Lop(AFL) [2], and the improvement mostly deal with integrating the fuzzer with the domain specific mutation engine. It relies on compile time instrumentation to make informed decision on which paths are interesting, and obtain the feedback in linear time,

and are thus applicable to large code base such as the AV platforms. However, it's also a promissing direction of future work to combine the domain specific mutation with selective symbolic or concolic executions to further improve the fuzzing practice.

4.4 The AutoFuzzer Approach

In this section, we first describe the scope of the problem , and discuss the limitations of the existing fuzzing tool when applied to these problems. We then introduce the major components of the AutoFuzzer approach, and present a roadmap for the detailed implementation of each component.

4.4.1 Problem Scope

As mentioned earlier, the major safety issue with appified AV comes from the large code base of the self-driving functionalities, and third party components. In this paper, we focus mainly on two types of issues – *performance issues* that breaks the real-time assumption of the AV runtime, such as the crashes or hangs of the program, and *vulnerabilities*, which have been consistently presented difficulties for other platforms. The design goal of AutoFuzzer is to efficiently detect as much unique crashes and hangs in the targeted AV platform, as they will bring safety risks, and some crashes that can be controlled by attacker input may also enable targeted attacks, such as remote control or denial of service. We consider two attack surfaces in this work: (1) Manipulation of sensor input and (2) Compromised in-vehicle modules.

We use the Baidu Apollo platform as shown in Figure 4.8 to illustrate the attack surface and our design. The essential hardware components including the GPS/IMU, camera, lidar, radar are installed on the drive-by-wire vehicle hardware platform, and communicate through the physical CAN bus. A real time operating system (RTOS) runs on the computing unit and acts as a middleware to provide the sensor data to the software modules that work together to accomplish self-driving tasks. In the Apollo platform, the GPS data from

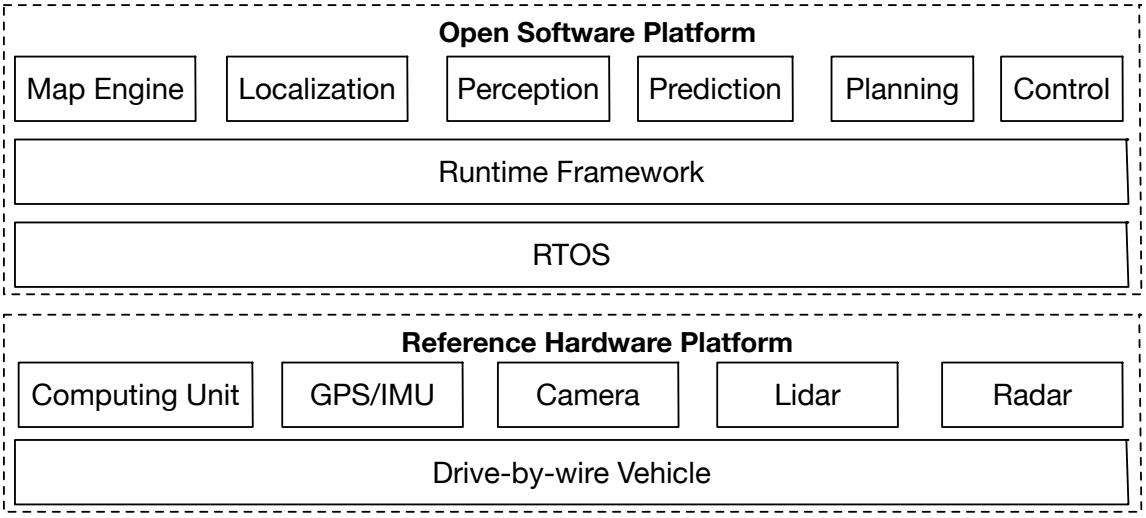


Figure 4.8: Apollo self-driving platform architecture

the IMU and the point cloud data from lidar are processed by the localization module and perception module respectively before sending to the prediction module. The prediction module receives the obstacles information including positions, velocities, accelerations, ect., and generates the predicted trajectories with the probabilities of the obstacles. Given the localization, vehicle status, map, predicted obstacles' status, the planning module will compute a trajectory that is safe and comfortable for controller to execute. At last. using different control algorithms, the control module generates the commands to be published to the steering wheel, throttle and brake to accomplish the self-driving task. In this work, we assume that the attacker could influence the decisions made by the AV or DoS certain critical functionalities either by perturbing the sensor input (e.g., lidar, camera) or by compromising certain module to manipulate the data sent to other modules. Each module itself is fault tolerant to some common problems, such as messages in bad format, or out-of-order messages. However, attackers can still targets on the embedded flaws to cause crash or hang of certain module to raise safety threats, or even compromise the whole system leveraging memory leaks.

4.4.2 Fuzzing

Fuzzing is a technique that executes an application with a wide set of inputs, checking if these inputs cause the application to crash. To retain speed of execution, fuzzers are minimally invasive, they perform minimal instrumentation on the underlying application and monitor it either from outside or inside. We will detail how AutoFuzzer improves the performance of existing fuzzing practice.

To implement AutoFuzzer, we leverage a popular off-the-shelf fuzzer, AFL [2]. AFL relies on instrumentation to make informed decisions on which paths are interesting. This instrumentation can be either introduced at compilation time or via a modified QEMU [67]. We opted for the Clang++ based instrumentation, since our targeted modules in Apollo are open sourced. We list and describe the most important AFL features, mentioning how they are used by AutoFuzzer to overcome certain challenges.

- **Target instrumentation.** The instrumentation injected into compiled target program captures branch (edge) coverage, along with coarse branch-taken hit counts. These instructions injected into each basic block reveal the current status exploring the whole program.

- **Genetic fuzzing.** AFL carries out input generation through a genetic algorithm, mutating inputs according to genetics-inspired rules (transcription, insertion, etc.) and ranking them by a fitness function. For AFL, the fitness function is based on unique code coverage, i.e., triggering an execution path that is different than the paths triggered by other inputs.

- **State transition tracking.** AFL tracks the union of control flow transitions that it has seen from its input, as tuples of the source and destination basic blocks. Inputs are prioritized for "breeding" in the genetic algorithm based on their discovery of new control flow transitions, meaning that inputs which cause the application to execute in a different way get priority in the generation of future inputs.

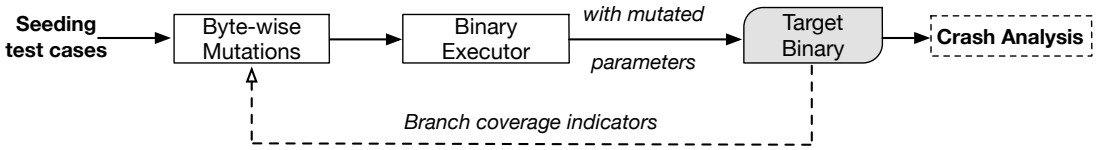


Figure 4.9: The workflow of AFL

- **Input mutation.** AFL supports a set of atomic mutations including bitwise flop, random subtraction/addition, insert bytes, clone bytes, overwrite bytes with a randomly selected chunk, etc. The feedback provided by the instrumentation makes it easy to understand the value of each fuzzing strategy and optimize their parameters to improve the efficiency.

However, AFL has certain limitations applying to the Apollo code base. First, AFL is designed to test target program that takes parameterized input either from the command line or from file, while on the AV platform, different modules work together in a distributed fashion, and communicate through their own IPC standard, we need to interface the fuzzer with the virtual CAN bus that connects these modules. Second, AFL can only target on single binary, which limits it to test only one AV module in the distributed system at a time. Systemic issues lying in the interactions between modules cannot be discovered, while false positives may occur, since erroneous input that crashes the target module may be filtered by previous modules. Third, differing from program that takes only parameterized input, the reliability of the stateful self-driving logic are also prone to the timing and sequence of messages arrivals. Simply fuzzing the target module by calling the message handler with parameterized input lose the timing and sequence information, and may result in high false negatives. At last, as mentioned before, the bytewise mutations is no longer effective in exploring deeper self-driving logic, and we are motivated to build AutoFuzzer to enhance the fuzzing practice of AFL.

4.4.3 AutoFuzzer Overview

AutoFuzzer is designed to discover crashes and hangs in the entire set of self-driving functionalities, including issues caused by the inter procedure communications among different processes. The bridge app connects AutoFuzzer to the virtual CAN bus, so that the mutated message can be published onto the bus, and consumed by target modules. To target critical AV functionalities as a whole instead of one at a time, AutoFuzzer mutate on the sequence of messages, which can be used to replay the road side condition during a period of time, and the the logic across all modules including planning, control will all be tested. The mutation engine of AutoFuzzer integrates domain specific atomic operations such as the insertion/removal/shift of obstacles. Compared with bytewise mutations, the domain specific mutations detects unique crashes and hangs in the autonomous driving functionalities more efficiently. We detail the implementation of the key components of AutoFuzzer in §4.4.4, and evaluate it effectiveness and efficiency in §4.5.

4.4.4 Design and Implementation

We prototype AutoFuzzer as a extensible fuzzing tool on the Apollo AV platform, which builds its self-driving modules as individual node on top of the Robots Operating System (ROS). ROS is not a Real Time Operating System (RTOS) by default since it uses Linux as its kernel, and it is inherently best-efforts in many cases, thus does not provide guarantee about the timing of operations. Apollo enables the real time support of the Linux kernel and relies on ROS to provide the best-efforts real time guarantee for critical tasks. The communication among Apollo modules goes through the ROS IPC , which uses socket I/O and shared memory as its underlying data transportation mechanism.

Bridge app is developed as a ROS node that implements the replay functionality of a sequence of messages. It publishs the mutated message sequence including a number of perception, localization and chassis messages that represent the roadside conditions during a period of time to the target Apollo modules. The replay mechanism also maintains the

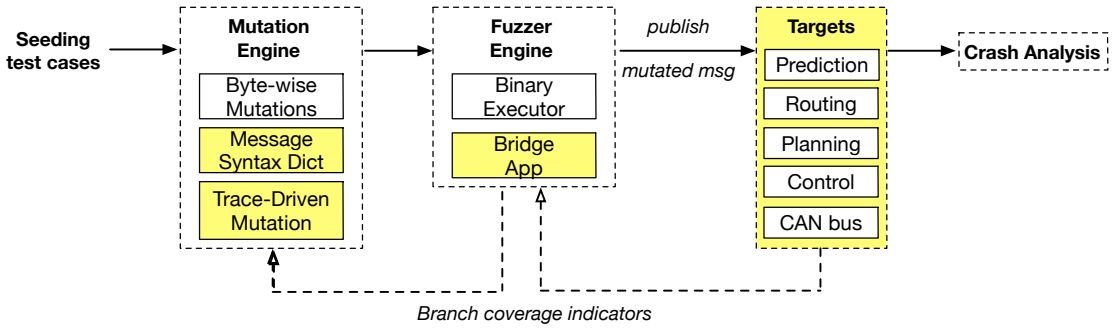


Figure 4.10: AutoFuzzer design

timing, sequence information, and inter-arrival delay of the message sequence to further broaden the scope of the tested logic. In addition, the bridger app captures the feedback from the instrumented target modules by monitoring their `pids`, and thus provides richer information to the front-end mutation engine than from single binary.

Message syntax dictionary is used to improve the efficiency of mutation, so that the generated test cases almost always make sense to the message handler, and thus explore deeper logic. By default, The mutation engine of AFL is optimized for compact data formats such as images, multimedia, compressed data. It is less suited for languages with particularly verbose and redundant syntax or human-readable language (e.g., RTF, HTML, JSON). For example, it's never easy to get from `trajectory_point{x:586.1}` to `header{module_name:"planning"}` by randomly flipping bits. AutoFuzzer provides a message syntax dictionary to the mutation engine, which gives certain atomic mutations such as inserting bytes, cloning bytes, and replacing bytes limited options, so that most messages generated by the mutation engine will not be captured by the bad format exception handlers in target modules.

Trace-driven mutation enables AutoFuzzer to fuzz all safety critical modules together rather than targets on one at a time. It performs mutations on a sequence of messages, which are all in the Apollo message format. However due to the real-world constraints in the self-driving scenario, such as the shape of the obstacles should be valid, the position of the obstacle should be within the lane, randomly mutating on the messages text can barely pass

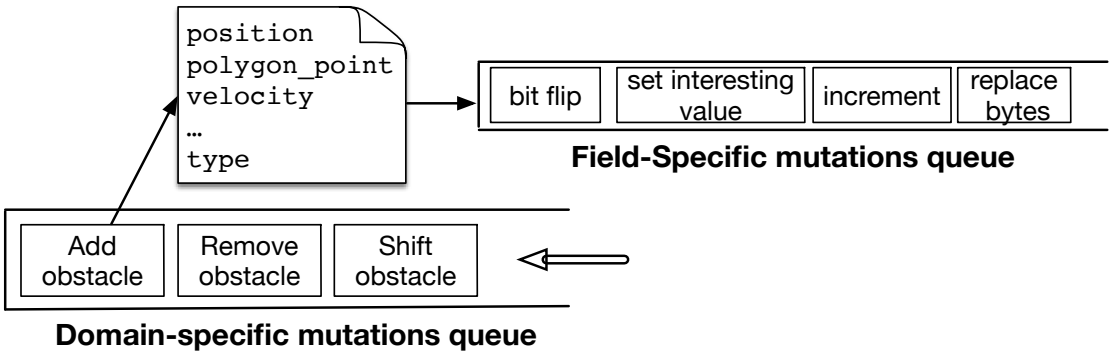


Figure 4.11: The mutation engine of AutoFuzzer

even shallow input validation checks. To address this issue, we propose adding domain-specific mutations to the fuzzing engine as shown in Figure 4.11. The basic idea is to translate realistic events that can happen in self-driving scenarios into atomic mutations that can be used by fuzzing. A unique `transition` stands for a unique tuple containing a source basic block and a destination basic block. Based on this feedback from the instrumented program, AFL chooses the most promising bytewise automic mutations to put in the action queue based on a fitness function. AutoFuzzer introduces a domain-specific mutations queue which uses the same value function to rank the mutations on obstacle level, so that mutated input can be easily interpreted to realistic events, and are highly likely to bypass shallow sanity checks. Meanwhile, the AutoFuzzer maintains a field-specific mutation queue for each domain-specific mutations to mutate the detail of obstacles with per field mutations under constraints.

Figure 4.12 shows how the targeted modules react to the obstacle inserted by the mutations in the simulator. Since Apollo 2.0 currently only tackles scenarios in simple urban road conditions [12], such as collision avoidance, we currently only implement three different types of domain-specific mutations – add, remove, and shift of obstacles. Our implementation can be easily extended to include more user-defined atomic mutations such as lane changes, traffic light changes, or even adversarial mutations in the future.

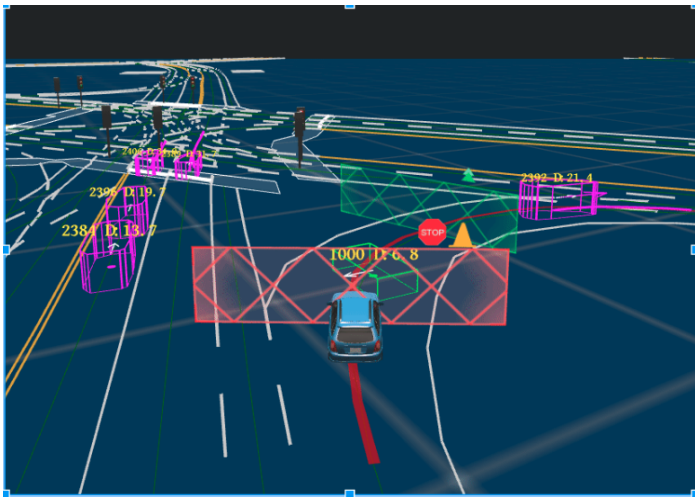


Figure 4.12: Running the mutated trace with inserted obstacles in our simulation

4.5 Evaluation

We evaluate the efficiency of AutoFuzzer on a Ubuntu 16.04 server machine with 8-core Intel Xeon 3.50GHz CPUs and 16 GB memory. Since the existing AFL cannot be directly compared with AutoFuzzer due to the lack of communication interface with the Apollo modules, and can only test one target binary at a time. To evaluate the improvement on fuzzing efficiency brought by the mutation engine, we built a baseline version of AFL that integrates the bridge app and the mutations on raw text of message sequences for fair comparison. We use the unique hangs and crashes produced by different fuzzing approaches over time as the performance metrics. The uniqueness are defined by AFL as hangs and crashes whose associated execution paths involve any state transitions not seen in previously-recorded faults. If a single bug can be reached in multiple ways, there will be some count inflation early in the process.

Shown in Figure 4.13 and Figure 4.14, AutoFuzzer produces unique hangs and crashes much more efficient than the baseline AFL approach given the same seeding test cases, which is a tailored message sequence provided by Apollo for demo purpose. As shown in the graph, when AFL fuzzes these the message handlers in targeted modules, the format checks cause the fuzzer to get stuck at the beginning and it takes about 6 hours to pro-

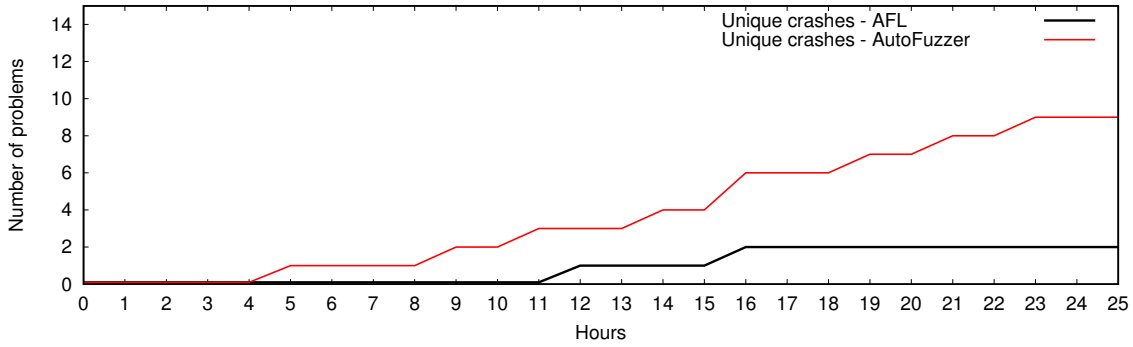


Figure 4.13: Produced unique crashes over time

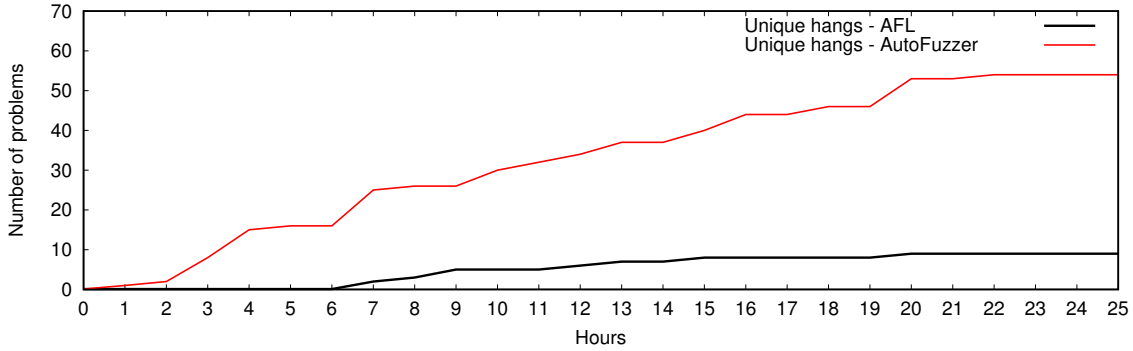


Figure 4.14: Produced unique hangs over time

duce the first hang, while AutoFuzzer almost immediately produces the hang after starting fuzzing leveraging the domain-specific atomic mutations.

4.6 Discussion and Future Work

In this section, we discuss the current limitation with our approach, and propose immediate and potential future work following this research direction.

4.6.1 Providing Root Cause Analysis Support

The major limitation of AutoFuzzer approach is the lack of support to efficiently perform root cause analysis on the unique crashes and hangs produced, due to the very limited information provided by the instrumentation. The common practice for the diagnosis is to start with the input that causes failures and use debugging tools such as `gdb` to attach to the targets and proceed step by step while monitoring the runtime behaviors. However, the diagnosis of some issues may require domain knowledge deep into the control algorithms being used, and we leave it as future work about how to integrates more domain knowledge and external exploitable verification techniques [22] into the diagnosis process.

4.6.2 Extending domain-specific mutations

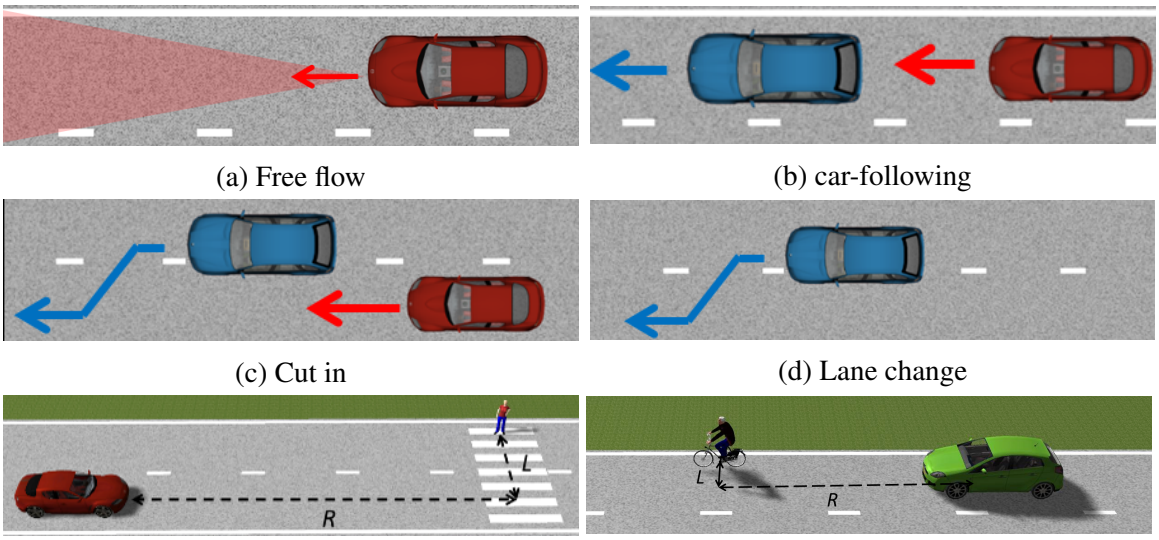
Another direction of further improving the AutoFuzzer approach is to add more domain-specific atomic operations into the mutation engine, thus producing more events in realistic driving scenarios. Recent work [138, 149] on the testing of the robustness of deep neural network has proposed the self-driving as one scenario that their neuron coverage aware fuzzing could apply. Although they have provided different mutation approaches such as blurring the camera image, flipping pixels of the image, their approach only works on testing certain self-driving tasks, such as the lane detection. We can explore the space of domain-specific constraints in the other sensors such as lidar and radar, and extend our atomic mutation set.

4.6.3 Naturalistic trace driven testing

In our domain-specific mutations, we assume the adversary has arbitrary control over the road side conditions, such as position and shape of obstacles, while another interesting problem is to evaluate how well the self-driving functionalities perform under normal naturalistic driving scenarios. A Naturalistic-Field Operational Test (N-FOT), data is collected from a number of on-board vehicles driven in naturalistic conditions over an extended period of time [62]. Many large-scale N-FOT projects have been conducted across U.S. For example the 100-Car Naturalistic Driving study [130] conducted by the Virginia Polytechnic Institute and State University studies the main factors causing vehicle crashes. While more recently, the Safety Pilot Model Deployment (SPMD) [134] launched by the University of Michigan Transportation Research Institute (UMTRI) was a comprehensive data collection effort under real-world conditions, with multimodal traffic and vehicles equipped with vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communication devices. The deployment included approximately 2,800 equipped vehicles and 30 roadside equipment. The data was logged and available in text format. These data will greatly help the fuzzing and other dynamic analysis approaches to test AV under naturalistic scenarios.

However, all the traffic data are logged and organized in the chronological order, and requires extensive post-processing to extract useful information such as the different scenarios where certain self-driving functionality would take control. Moreover, these datasets contain the raw sensor data, collected from on-board equipment such as radar, Mobileye, which usually results in extremely large file size, and thus raises the bar for vehicle engineer and researchers without the big data analytics background, or sufficient computing resources to utilize them.We believe that a well-organized and maintained scenarios library will address the usability issue, and significantly increase the AV development efficiency.

We have taken some initial efforts towards this goal by building a library for self-driving scenarios, called TrafficNet[48] to reduce efforts for to use the open datasets, and thus bridge the gap between research datasets and practically usable information for vehicle en-



(a) Free flow (b) car-following

(c) Cut in (d) Lane change

(e) Pedestrian (f) Cyclist

Figure 4.15: Scenarios in TrafficNet

gineers and researchers. It labels the data collected from many open dataset and categorize and label the data with statistical analysis into six key driving scenarios shown in Figure 4.15. Both the extracted events, raw data, and source code are provided online for free access. TrafficNet provides scenario-based structure rather than chronological raw data, which helps the researchers to test AV functionalities more efficiently. We also supports the crowdsourcing development of the algorithms to label more critical driving scenarios for finer-grained testing.

4.7 Conclusion

We design AutoFuzzer as a specialized fuzzing solution for the emerging AV platform, and prototype it on the Baidu Apollo [12], which is yet the most mature AV platform that is publicly available. Specifically AutoFuzzer introduces an enhanced mutation engine that includes realistic atomic perturbations to improve the efficiency of fuzzing, and addresses the challenge of testing the distributed in-vehicle system by interfacing the fuzzer with the internal Inter Procedure Communication (IPC) mechanism. AutoFuzzer provides a portable solution without requiring any change to the target code bases, and is gener-

ally applicable to AV platforms with the similar ROS based architecture. The evaluation shows that it detects unique crashes and hangs much more efficient than the state-of-the-art fuzzing tool in the self-driving scenarios.

CHAPTER V

Future Work & Conclusion

In this chapter, we look at future research directions based on this research and conclude by highlighting the contributions presented in this dissertation.

5.1 Future Work

5.1.1 Usability Research on Context-based Access Control

In our ContexIoT work, we prototyped a novel context-based access control system for appified IoT platform, while our focus is mainly on the completeness and soundness of context collection when sensitive actions are triggered, the usability evaluation, regarding how well user can make the permission granting decision based on the presented context description. We think it's equally important to ensure that the rich context collected by ContexIoT is presented in a way that can be well perceived by the IoT users. The current text based presentation of context can be improved in many ways. For example, leveraging the trigger-action based programming model of IoT apps, the context can be presented in a graphic event chain to make it clear how the sensitive actions is triggered by a series of events, or under a set of conditions.

Another direction for improving usability is to take human out of the loop through automatic generation of policy based on the app logic and its interactions with user. For example, the SmartApp has a setup procedure during the app installation, which requires

user to set some parameters that will guide the automation of the SmartApp (e.g., automatically locks the door when it has been opened for 2 minutes). Using the data-dependency tracking support of ContexIoT, the security logic can monitor how the app uses the user input. If it detects that the events corresponding to the unmodified user input triggers the user's desired action at the runtime, the execution can be automatically allowed since it conforms with the user specified routine. Natural Language Processing (NLP) technique may be required to infer the user desired action. We leave it as future work to explore these extended usages of the ContexIoT.

5.1.2 Fuzz Testing for Self-driving Functionality with Deep Learning Components

The self-driving functionalities targeted by AutoFuzzer don't have the deep learning based components yet. Since recent advances in Deep Neural Network (DNN) have led to the development of DNN-driven autonomous cars that combines classic perception, planning, and control algorithms with DNN based obstacle detection and classification, it's also critical to adapt the fuzz testing approach to these modules. Although the approaches for testing the reliability of DNN have been propose [138, 149], their targeted use cases are pretty simple, and is more like a toy application compared with the AV platform such as Apollo. Moreover, it's still a open question how to test a compound program that combines DNN and traditional algorithms with guided fuzzing. The fuzzing of traditional software can be guided with code coverage, while the testing of DNN models can be guided with neuron coverage, and we propose a uniformed approach as future work, which uses a global fitness function to find the most valuable mutation. To find the optimal solution for the fitness function, we will use the gradient descent on both DNN modules and traditional algorithms, which requires the regression for these algorithms, and we leave it as the future work.

5.1.3 Verification Support for Self-driving Functionality

Formal verification can be helpful in proving the correctness of system such as cryptographic protocols, combinational circuits, etc. The verification is down by providing a formal proof on an abstract mathematical model of the system, and previous work [116, 135], has shown that it is feasible to check that whether certain property can be satisfied throughout the lifecycle of a program. However, as DNN is now being deployed as controllers for safety-critical system such as the autonomous vehicles, it's a promising direction to also provide formal guarantees about the behavior of DNN-involved self-driving functionalities. Verifying DNNs is a difficult problem, since they are large, non-linear, and non-convex, and verifying even simple properties about them is an NP-complete problem. Fortunately, recent efforts has demonstrated that it is feasible to verify certain properties of specific types of DNNs using specialized SMT solvers [113], such as that small perturbations do not change the advisories produced for certain inputs. However, these approaches only work with DNNs with ReLU activation functions, and a promising direction for future work would be to further extend the simplex algorithm to support other non-linear non-convex activation functions to better support the verification of self-driving functionalities.

5.2 Concluding Remarks

Despite the benefit of providing an enriched set of functionalities, the appification of platforms also comes with security and safety risks by allowing untrusted third party code to control user's device. However,in reality, the threats from the vulnerable and malicious apps have never been thoroughly mitigated even on the most mature personal computing device – the smartphone, not to mention the emerging smart home and autonomous vehicle platforms.

The dissertation proposes practical solutions towards addressing the aforementioned problems. More specifically, my research try to advance the security of the platforms in

three emerging domains by identifying design and implementation flaws in programs that lead to attacks and safety threats, and build platform-level defense to mitigate the threats. Utilizing static and dynamic program analysis, I 1) systematically examined the vulnerabilities exposed by open port usage on mobile devices which lead to remote large-scale attacks; 2) designed and implemented a new access control model for appified IoT platform that enforces contextual integrity; 3) proposed a holistic approach to enhance the security and safety open autonomous vehicle platform. In summary, my dissertation demonstrates that: Systematic program analyses of software (1) Lead to an understanding of design and implementation flaws across different platforms that can be leveraged in miscellaneous attacks or causing safety problems; (2) Lead to the development of security mechanisms that limit the potential for these attacks.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Active X Exploitation. http://resources.infosecinstitute.com/active-x-exploitation/#gref.

[2] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[3] Android 5.0, Lollipop. https://www.android.com/versions/lollipop-5-0/.

[4] Android 6.0, Marshmallow. https://www.android.com/versions/marshmallow-6-0/.

[5] Android Things. https://developer.android.com/things/index.html.

[6] Android Universally Unique Identifier. http://developer.android.com/reference/java/util/UUID.html.

[7] Anzhi app market. http://www.anzhi.com.

[8] Apple HomeKit. https://developer.apple.com/homekit/.

[9] ARM processor specifications. https://www.hex-rays.com/products/ida/support/idadoc/1350.shtml.

[10] Arp-scan. http://linux.die.net/man/1/arp-scan.

[11] AutoLockDoor SmartApp. https://github.com/smartthings-users/smartapp.auto-lock-door/blob/master/auto-lock-door.smartapp.groovy.

[12] Baidu Apollo Project. http://apollo.auto/.

[13] Car hacking: how big is the threat to self-driving cars? http://fortune.com/2014/10/07/car-hacking-how-big-is-the-data-threat-to-self-driving-cars/.

[14] China-Made Handheld Barcode Scanners Ship with Spyware. http://www.tomsguide.com/us/chinese-barcode-scanner-spyware,news-19157.html.

[15] Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.

[16] Compile-time Metaprogramming. http://groovy-lang.org/metaprogramming.html.

[17] Control Area Network (CAN). http://www.ni.com/white-paper/2732/en/.

[18] CVE-2015-6602. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6602.

[19] CWE-676:Use of Potentially Dangerous Functions. https://cwe.mitre.org/data/definitions/676.html.

[20] Diffie-Hellman Key Agreement. https://www.ietf.org/rfc/rfc2631.txt.

[21] Epoll I/O notification. http:linux.die.net/man/4/epoll.

[22] GDB Exploitable Plugin. https://github.com/jfoote/exploitable.

[23] Google Weave Project. https://developers.google.com/weave/.

[24] Hackers Remotely Kill a Jeep on the Highway. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[25] Heartbleed. http://heartbleed.com/.

[26] IDA-Pro. https://www.hex-rays.com/index.shtml.

[27] Inside the Self-Driving Tesla Fatal Accident. http://www.nytimes.com/interactive/2016/07/01/business/inside-tesla-accident.html.

[28] iOS 9. http://www.apple.com/ios/.

[29] MCity, University of Michigan. https://mcity.umich.edu/.

[30] Meltdown and Spectre. https://meltdownattack.com/.

[31] Mirai Attacks. https://securityledger.com/2016/11/report-millions-and-millions-of-devices-vulnerable-in-latest-mirai-attacks/.

[32] Mobile Open Port Security Project. https://sites.google.com/site/openportsec.

[33] Mobileye Technology. https://www.mobileye.com/en-us/technology/.

[34] Nest Cam Indoor. https://nest.com/camera/meet-nest-cam/.

[35] One hundred days before and after Baidu Wormhole's Discovery. http://www.inforsec.org/wp/wp-content/uploads/2016/01/\wormhole_external_final.pdf.

[36] OpenXC Platform. http://openxcplatform.com/.

[37] Our Project Website. https://sites.google.com/site/iotcontextualintegrity/home.

[38] PolySync. https://polysync.io/.

[39] Researchers exploit ZigBee security flaws that compromise security of smart homes. http://www.networkworld.com/article/2969402/microsoft-subnet/researchers-exploit-zigbee-security-flaws-that-compromise-security-of-smart-homes.html.

[40] Robot Operating System. http://www.ros.org/.

[41] Samsung SmartThings. https://www.smartthings.com.

[42] Schlage Z-Wave Lock. http://www.schlage.com/en/home/products/products-connected-devices.html.

[43] Smartisan technology. http://www.smartisan.com.

[44] SmartThings Compatible Products. https://www.smartthings.com/compatible-products.

[45] SmartThings Simulator. http://docs.smartthings.com/en/latest/device-type-developers-guide/simulator-metadata.html.

[46] Surveillance cameras sold on Amazon infected with malware. http://www.zdnet.com/article/amazon-surveillance-cameras-infected-with-malware/.

[47] The Conficker Worm. https://www2.sans.org/security-resources/malwarefaq/conficker-worm.php.

[48] The TrafficNet Project. http://traffic-net.org/.

[49] U-M Open-access Self-driving Vehicle. http://ns.umich.edu/new/releases/24351-u-m-offers-open-access-automated-cars-to-advance-driverless-research.

[50] UPnP: Universal Plug and Play. https://tools.ietf.org/html/rfc6970.

[51] US-CERT. https://www.us-cert.gov/.

[52] Velodyne Lidar. http://velodynelidar.com/.

[53] Windows Access Control Overview. https://docs.microsoft.com/en-us/windows/security/identity\discretionary{-}{}{}protection/access\discretionary{-}{}{}control/access\discretionary{-}{}{}control.

[54] Xposed Module Repository. http://repo.xposed.info/.

[55] ZOOX. http://zoox.com/.

[56] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *ACM CCS*. ACM, 2015.

[57] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. Sok: Lessons learned from android security research for appified software platforms. In *2016 IEEE Symposium on Security and Privacy*, 2016.

[58] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[59] A. Albert. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. *Embedded World*, 2004:235–252, 2004.

[60] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*. ACM, 2014.

[61] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[62] M. Aust. Evaluation process for active safety functions: Addressing key challenges in functional, formative evaluation of advanced driver assistance systems. *Chalmers University of Technology*, 2012.

[63] S. Babar, A. Stango, N. Prasad, J. Sen, and R. Prasad. Proposed embedded security framework for internet of things (iot). In *Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference on*, pages 1–5. IEEE, 2011.

[64] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016.

[65] X. Bai, L. Xing, N. Zhang, X. Wang, X. Liao, T. Li, and S.-M. Hu. Staying secure and unprepared: Understanding and mitigating the security risks of apple zeroconf. 2016.

[66] M. Ballano. Android Threats Getting Steamy. `https://www.symantec.com/connect/blogs/android-threats-getting-steamy`.

[67] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[68] P. Boonstoppel, C. Cadar, and D. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366. Springer, 2008.

[69] M. Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 33–42. ACM, 2006.

[70] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 131–146, 2013.

[71] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, and D. Song. Transformation-aware exploit generation using a hi-cfg. Technical report, California Univ Berkeley Dept of Electrical Engineering and Computer Science, 2013.

[72] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 725–741. IEEE, 2015.

[73] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security*, 2015.

[74] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou. Following devils footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *2016 IEEE Symposium on Security and Privacy*, 2016.

[75] Q. A. Chen, Z. Qian, Y. Jia, Y. Shao, and Z. M. Mao. Static Detection of Packet Injection Vulnerabilities – A Case for Identifying Attacker-controlled Implicit Information Leaks. In *ACM CCS*, 2015.

[76] Q. A. Chen, Z. Qian, Y. J. Jia, Y. Shao, and Z. M. Mao. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 388–400. ACM, 2015.

[77] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security*, 2014.

[78] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[79] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *ACM Mobisys*, 2011.

[80] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In *International Conference on Information Security*, pages 331–345. Springer, 2010.

[81] I. D. Cooperation. Q4, 2012 Smartphone Market shares- IDC. `https://ronnie05.wordpress.com/2013/02/15/q4-2012-smartphone-market-shares-idc/`.

[82] S. Demetriou, X.-y. Zhou, M. Naveed, Y. Lee, K. Yuan, X. Wang, and C. A. Gunter. What's in your dongle and bank account? mandatory and discretionary protection of android external resources. In *NDSS*, 2015.

[83] W. Diao, X. Liu, Z. Li, and K. Zhang. No pardon for the interruption: New inference attacks on android through interrupt timing analysis. In *2016 IEEE Symposium on Security and Privacy*. IEEE, 2016.

[84] W. Drewry and T. Ormandy. Flayer: Exposing application internals. *WOOT*, 7:1–9, 2007.

[85] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *USENIX Security*, 2013.

[86] W. M. Eddy. TCP SYN Flooding Attacks and Common Mitigations. rfc4987, 2007.

[87] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for real-time privacy monitoring on smartphones. *ACM TOCS*, 32(2):5, 2014.

[88] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *ACM CCS*, 2012.

[89] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, D. Wagner, et al. How to ask for permission. In *HotSec*, 2012.

[90] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-delegation: Attacks and Defenses. In *USENIX Security Symposium*, 2011.

[91] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. Android UI Deception Revisited: Attacks and Defenses. In *FC*, 2016.

[92] E. Fernandes, J. Jung, and A. Prakash. Security Analysis of Emerging Smart Home Applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, May 2016.

[93] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proceedings of the 25th USENIX Security Symposium*, August 2016.

[94] B. Fouladi and S. Ghanoun. Honey, im home!!-hacking z-wave home automation systems. *Black Hat*, 2013.

[95] A. Francillon, B. Danev, and S. Capkun. Relay attacks on passive keyless entry and start systems in modern cars. In *NDSS*, 2011.

[96] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In *IEEE Security & Privacy*, 2016.

[97] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *TRUST*, 2012.

[98] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[99] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[100] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow Analysis of Android Applications in DroidSafe. In *NDSS*, 2015.

[101] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[102] W. G. Halfond, J. Viegas, A. Orso, et al. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[103] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, pages 49–64, 2013.

[104] S. Heelan. Vulnerability detection systems: Think cyborg, not robot. *IEEE Security & Privacy*, 9(3):74–77, 2011.

[105] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 461–472. ACM, 2016.

[106] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[107] W. Hu, D. Octeau, P. D. McDaniel, and P. Liu. Duet: library integrity verification for android applications. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 141–152. ACM, 2014.

[108] H. Huang, S. Zhu, K. Chen, and P. Liu. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[109] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In *TRUST*. 2013.

[110] S. Jero, E. Hoque, D. Choffnes, A. Mislove, and C. Nita-Rotaru. Automated attack discovery in tcp congestion control using a model-guided approach. 2018.

[111] Y. Z. X. Jiang. Detecting passive content leaks and pollution in android applications. In *NDSS*, 2013.

[112] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[113] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[114] P. G. Kelley, L. F. Cranor, and N. Sadeh. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2013.

[115] A. Kharraz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda. Unveil: A large-scale, automated approach to detecting ransomware.

[116] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[117] P. Lantz and B. Johansson. Towards bridging the gap between dalvik bytecode and native code during static analysis of android applications. In *Wireless Communications and Mobile Computing Conference*, 2015.

[118] K.-S. Lhee and S. J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, 33(5):423–460, 2003.

[119] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

[120] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 978–989. ACM, 2014.

[121] C.-C. Lin, H. Li, X.-y. Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *NDSS*, 2014.

[122] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *ACM CCS*, 2012.

[123] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee. The price of free: Privacy leakage in personalized mobile in-app ads. 2016.

[124] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[125] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security & Privacy*, 2003.

[126] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. 2018.

[127] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. Uipicker: User-input privacy identification in mobile applications. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[128] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[129] M. Naveed, X.-y. Zhou, S. Demetriou, X. Wang, and C. A. Gunter. Inside job: Understanding and mitigating the threat of external device mis-binding on android. In *NDSS*, 2014.

[130] V. L. Neale, T. A. Dingus, S. G. Klauer, J. Sudweeks, and M. Goodman. An overview of the 100-car naturalistic study and findings. *National Highway Traffic Safety Administration, Paper*, (05-0400), 2005.

[131] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 87–97. ACM, 2015.

[132] H. Nissenbaum. Privacy as contextual integrity. *Wash. L. Rev.*, 79:119, 2004.

[133] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, 2013.

[134] U. D. of Transportation. Safety pilot model deployment data. `https://www.its.dot.gov/factsheets/safetypilot_modeldeployment.htm`.

[135] S. Owre, J. Rushby, N. Shankar, and F. Von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[136] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow. Iotpot: analysing the rise of iot compromises. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[137] B. S. Pak. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *School of Computer Science Carnegie Mellon University*, 2012.

[138] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.

[139] A. Rahmati and H. V. Madhyastha. Context-specific access control: Conforming permissions with user expectations. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '15, 2015.

[140] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[141] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *ACM ASIACCS*, 2013.

[142] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[143] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *2012 IEEE Symposium on Security and Privacy*, pages 224–238. IEEE, 2012.

[144] E. Ronen and A. Shamir. Extended functionality attacks on iot devices: The case of smart lights. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 3–12. IEEE, 2016.

[145] J. Seo, D. Kim, D. Cho, T. Kim, and I. Shin. Flexdroid: Enforcing in-app privilege separation in android. 2016.

[146] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *NDSS*, 2016.

[147] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[148] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[149] Y. Tian, K. Pei, S. Jana, and B. Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. *arXiv preprint arXiv:1708.08559*, 2017.

[150] B. Ur, J. Jung, and S. Schechter. The current state of access control for smart devices in homes. In *Workshop on Home Usable Privacy and Security (HUPS)*, 2013.

[151] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014.

[152] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *ACM SIGMETRICS Performance Evaluation Review*, 2014.

[153] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 635–646. ACM, 2013.

[154] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 497–512. IEEE, 2010.

[155] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Citeseer, 2009.

[156] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *ACM CCS*, 2014.

[157] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, 2015.

[158] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[159] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[160] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[161] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 303–313. IEEE, 2015.

[162] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 5. ACM, 2015.

[163] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen. Sidebuster: Automated Detection and Quantification of Side-channel Leaks in Web Application Development. In *CCS*, 2010.

[164] L. Zhang, M. Gordon, R. Dick, Z. M. Mao, P. Dinda, and L. Yang. ADEL: An Automatic Detector of Energy Leaks for Smartphone Applications. In *CODESISSS*, 2012.

[165] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on android. In *2015 IEEE Symposium on Security and Privacy*, pages 915–930. IEEE, 2015.

[166] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du. Life after app uninstallation: Are the data still alive? data residue attacks on android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, California, USA*, 2016.

[167] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1017–1028. ACM, 2013.

[168] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423. IEEE, 2014.

[169] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, 2012.

[170] Y. Zhou and X. Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *NDSS*, 2013.

[171] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS*, 2012.